

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



Hyperledger代码贡献者撰写，Hyperledger亚太区副总裁Julian Goldon、ChinaLedger技术委员会主任白硕以及多个区块链公司的CEO、CTO、科学家联袂推荐  
零基础掌握Hyperledger Fabric的关键技术、工作原理和应用开发方法，多个案例实战并附项目源代码

BLOCKCHAIN IN ACTION  
Key Technology and Case Analysis for Hyperledger Fabric

# 区块链开发实战

Hyperledger Fabric关键技术与案例分析

冯翔 刘涛 吴寿鹤 周广益 ◎著



机械工业出版社  
China Machine Press

## 内容简介

本书是“区块链开发实战”系列的第1本，旨在让零基础的读者也能迅速掌握 Hyperledger Fabric 的各种基本概念、关键技术、工作原理、应用开发方法。作者是国内区块链领域的早期实践者和布道者，Hyperledger 核心项目的核心开发者，在区块链技术开发领域积累了丰富的项目经验。本书得到了 ChinaLedger 技术委员会主任白硕、MATRIX 区块链首席 AI 科学家邓仰东、阿希链 CTO 钱汉涛、元界 CEO 陈浩等多位专家的鼎力推荐。

全书主要内容在逻辑上分为三个部分：

### 第一部分 准备篇（第 1~2 章）

这部分介绍了从事区块链开发需要具备的预备知识，如区块链的各种概念、开发环境的搭建和开发工具的使用等。

### 第二部分 以太坊篇（第 3~13 章）

这部分是本书的核心内容，系统、全面地讲解和分析了 Hyperledger Fabric 的各种基本概念、关键技术、工作原理，以及应用开发方法。如 Hyperledger 的技术体系，以及 Hyperledger Fabric 的基本概念、核心模块、账号体系、智能合约、编程接口、系统架构设计、应用开发流程。除此之外，还有区块链浏览器、供应链金融和食品溯源方面的 3 个综合案例。

### 第三部分 扩展篇（附录）

详细介绍了比特币的工作原理、运行方式、功能模块、编程接口，以及基于比特币的应用开发方法，对于想研究比特币技术原理和从事比特币应用开发的读者来说，是一份难得的资料。





华章IT  
HZBOOKS | Information Technology







区块链  
技术丛书

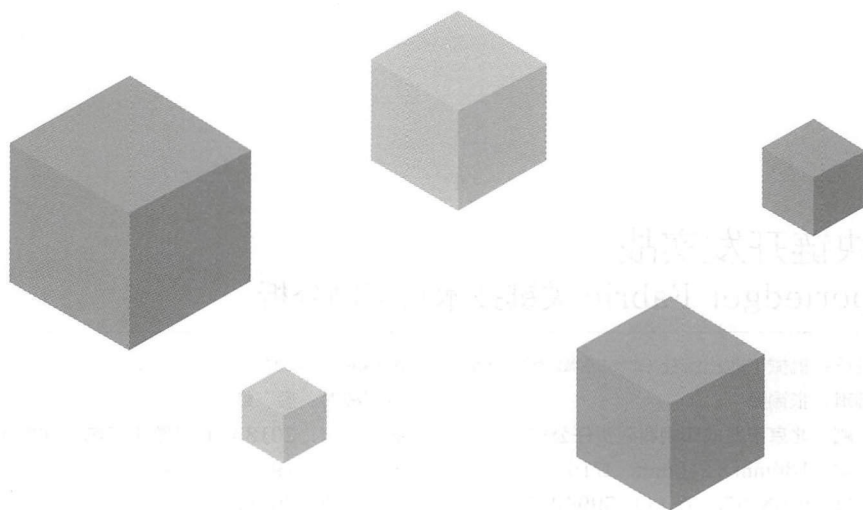
BLOCKCHAIN IN ACTION

Key Technology and Case Analysis for Hyperledger Fabric

# 区块链开发实战

Hyperledger Fabric关键技术与案例分析

冯翔 刘涛 吴寿鹤 周广益 ◎ 著



机械工业出版社  
China Machine Press



Visual Studio

书籍是人类进步的阶梯

## 图书在版编目 (CIP) 数据

区块链开发实战: Hyperledger Fabric 关键技术与案例分析 / 冯翔等著. —北京: 机械工业出版社, 2018.5

(区块链技术丛书)

ISBN 978-7-111-59942-5

I. 区… II. 冯… III. 电子商务—支付方式—研究 IV. F713.361.3

中国版本图书馆 CIP 数据核字 (2018) 第 095140 号

## 区块链开发实战

### Hyperledger Fabric 关键技术与案例分析

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 张锡鹏

责任校对: 殷虹

印刷: 北京市兆成印刷有限责任公司

版次: 2018 年 6 月第 1 版第 1 次印刷

开本: 186mm × 240mm 1/16

印张: 19.75

书号: ISBN 978-7-111-59942-5

定价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东



## Preface 前言

### 为何写作本书

近年来区块链技术逐步占据各大技术类网站的头条，各种基于区块链特性的想法和创新层出不穷。这些繁荣是区块链技术在幕后默默支撑的，可是人们经常忽略区块链的技术而把投资、融资、保值等金融属性和区块链画上了等号。其实区块链本质上还是一门技术。区块链技术源于比特币，经过近几年的发展，已经超越比特币逐步形成一门单独的技术体系。目前区块链技术已经渗透到各行各业中，比如区块链技术同大数据、人工智能等技术产生了让人意想不到的化学反应。我们有理由相信区块链技术在未来一定会成为 IT 基础技术之一，成为每个 IT 技术人员必备的基础技能。

同时我们也可以看到区块链技术在国内外的发展非常迅速。在国外，IBM 发起了超级账本项目，并把超级账本项目的源码捐献给了 Linux 基金，借助社区的力量来发展。全球已经有将近 200 多个公司和组织加入了超级账本，成为超级账本项目的会员。当然其他巨头也随之跟进，微软早就和以太坊达成了战略合作协议。互联网巨头 Google、社交媒体行业的龙头 Facebook 等在区块链领域均有所布局。

但是在繁荣的背后我们也应该看到危机，目前区块链技术在项目中的应用还存在不少问题。我们认为出现这种情况是因为目前区块链技术的实用化还存在以下障碍：

- 技术新，学习资料匮乏。区块链技术是最近几年刚刚兴起的一门综合技术，目前资料特别是中文资料还是比较缺乏的。
- 技术种类多，有一定的学习成本。区块链是一门综合型的技术，如果把每个单项技术列出来学习并不难，但是当把这些技术组合起来之后学习难度就大大增加了。
- 可借鉴的成功案例少。由于区块链技术是一门比较新的技术，因此目前缺少比较成功的案例。即使诸如 IBM 等巨头开发了一些成功案例，但是由于各种各样的原因，目



前并没有公开，这些都给广大技术人员学习区块链技术特别是把区块链技术应用到具体项目中造成了一定的障碍。

这些问题的存在是我们编写“区块链开发实战”系列图书的目的，第一批有两本书同时面世，分别是基于 Hyperledger Fabric 和以太坊进行区块链开发实战。我们希望读者通过这两本书，在了解区块链的基本概念和核心技术的同时，能够将区块链技术更多应用到具体的项目中，解决现有技术无法解决的一些行业痛点。

## 读者对象

这两本书都非常适合区块链开发工程师、区块链架构师、区块链技术爱好者阅读。

其中：

- Hyperledger Fabric 部分更适合对 Hyperledger Fabric 和比特币技术感兴趣的相关技术人员；
- 以太坊部分更适合以太坊爱好者、以太坊 DAPP 开发者、比特币开发者等。

## 主要内容

### 《区块链开发实战：Hyperledger Fabric 关键技术与案例分析》

这本书以 Hyperledger Fabric 和比特币这两个典型区块链技术平台的核心技术、开发方法和相关的项目案例为核心内容，此外，还提供了大量的命令脚本和代码示例供读者参考，力图使读者在最短的时间内掌握这两个平台的使用方法。

全书分为三个部分：

- 第一部分（第 1 ~ 2 章）：首先从基本认识的角度对区块链进行了宏观上的介绍，包括区块链技术的起源和演进过程、区块链核心技术及其特性、区块链技术的缺点和常见错误认识，以及区块链技术的应用领域和常见的技术框架；然后介绍了进行区块链开发需要掌握的技术和使用的工具。
- 第二部分（第 3 ~ 13 章）：主要讲解了 Hyperledger Fabric 的核心技术、原理、开发方法，以及多个项目案例。包括 Hyperledger 的全面介绍、Fabric 的技术特性和快速入门、Fabric 的核心模块和账号体系、Fabric 的智能合约和编程接口、Fabric 的系统架构与设计、Fabric 项目案例的开发流程和方法，以及几个综合性的案例，如区块链浏览器、供应链金融、食品溯源等。
- 第三部分（附录）：主要讲解了比特币的原理、运行方式、重要模块和编程接口，同时还讲解了一个比特币客户端的案例。





## 《区块链开发实战：以太坊关键技术与案例分析》

本书详细讲解了以太坊和比特币这两个典型的区块链技术平台的技术特性、原理、开发方法，同时也配有多个综合性的项目实例。

全书分为三个部分：

- 第一部分（第 1 ~ 2 章）：首先从基本认识的角度对区块链进行了宏观上的介绍，包括区块链技术的起源和演进过程、区块链核心技术及其特性、区块链技术的缺点和常见错误认识，以及区块链技术的应用领域和常见的技术框架；然后介绍了进行区块链开发需要掌握的技术和使用的工具。
- 第二部分（第 3 ~ 11 章）：主要讲解了以太坊的基本使用、技术特性、工作原理、开发方法和项目案例。首先介绍了以太坊的各种核心概念——编译、安装、运行，以及私有链的搭建和运行等基础内容；其次详细讲解了 Solidity 语法、Solidity IDE、Solidity 智能合约的编译部署，以及 Solidity 的智能合约框架 Truffle；最后讲解了 DApps 开发的方法和流程。
- 第三部分（附录及后记）：主要讲解了比特币的原理、运行方式、重要模块和编程接口，同时还讲解了一个比特币客户端的案例。

## 为什么两本书有重复内容

大家可能注意到，两本书有部分内容是重复的，这么安排并不是为了凑篇幅，而是经过精心考虑的。主要原因如下：

- 以太坊和 Hyperledger Fabric 是两个不同的技术平台，涉及的技术都非常多，读者一般不会同时学习并在这两个平台上进行开发，于是我们没有将这两个主题的内容放到一本书中，这样便于读者按需选择。
- 两本书的前两章是相同的，因为这两章的内容对两个平台的用户来说是通用的，而且是都需要了解和学习的。
- 两本书关于比特币的内容是相同的，因为比特币系统是出现最早、运行最稳定的区块链技术平台，它的很多概念和核心技术对其他区块链平台有非常好的借鉴意义，值得所有区块链开发者学习。

## 主要特色

这两本书是作者在参与众多区块链项目之后提炼而成，具有以下特点：

- 既没有高深的理论也没有晦涩难懂的公式，力求通过最简单通俗的语言和大量的图表让读者能够了解区块链技术的精髓。

- 提供大量的命令脚本和相关程序的源代码文件，这些命令脚本和源代码文件都来自实际的项目，我们整理后展现给读者，通过这些命令和源代码读者可以了解到相关区块链技术平台的操作细节。
- 提供了大量的项目案例，这些项目案例能够帮助读者更好地理解区块链技术和业务场景的结合。
- 与国内专业的区块链技术社区——“区块链兄弟”深度合作，社区中有两本书的专题页面，读者可以到社区中与作者和其他读者进行深入交流。

本书相关源代码下载地址：<https://github.com/blockchain-technical-practice>。

## 致谢

这本书能够完成首先要感谢机械工业出版社华章公司的杨福川先生为本书的顺利出版付出的努力。同时我们要感谢区块链技术社区的全体“兄弟”，你们对区块链的探索和执着是我们创作的动力，你们对区块链的付出和努力给我们提供了创作的素材。在编写这本书的过程中无论是提问题的“兄弟”，还是回答问题的专家“兄弟”，感谢你们。最后我们还要感谢所有加入的区块链技术讨论组，在和你们的交流中我们发现了本书的价值。

本书编写小组

2018年2月于上海



# Contents 目 录

## 前言

## 第 1 章 全面认识区块链 ..... 1

- 1.1 区块链技术的起源和解释 ..... 1
- 1.2 区块链的核心技术及其特性 ..... 2
  - 1.2.1 区块链技术的特性 ..... 3
  - 1.2.2 区块链的分布式存储技术特性 ..... 3
  - 1.2.3 区块链的密码学技术特性 ..... 4
  - 1.2.4 区块链中的共识机制 ..... 8
  - 1.2.5 区块链中的智能合约 ..... 12
- 1.3 区块链技术演进过程 ..... 13
- 1.4 区块链技术的 3 个缺点 ..... 13
- 1.5 区块链技术常见的 4 个错误认识 ..... 14
- 1.6 区块链技术的应用领域 ..... 15
  - 1.6.1 区块链在金融行业的应用 ..... 15
  - 1.6.2 区块链在供应链中的应用 ..... 16
  - 1.6.3 区块链在公证领域的应用 ..... 17
  - 1.6.4 区块链在数字版权领域的应用 ..... 18
  - 1.6.5 区块链在保险行业的应用 ..... 19
  - 1.6.6 区块链在公益慈善领域的应用 ..... 21
  - 1.6.7 区块链与智能制造 ..... 22
  - 1.6.8 区块链在教育就业中的应用 ..... 23

## 1.7 区块链的其他常见技术框架 ..... 24

## 1.8 本章小结 ..... 25

## 第 2 章 实战准备 ..... 26

- 2.1 开发环境准备 ..... 26
  - 2.1.1 操作系统的配置 ..... 26
  - 2.1.2 Docker 的使用 ..... 27
  - 2.1.3 Git 的使用 ..... 30
- 2.2 开发语言 ..... 30
  - 2.2.1 GO 语言 ..... 30
  - 2.2.2 Node.js ..... 32
- 2.3 常用工具 ..... 32
  - 2.3.1 Curl ..... 32
  - 2.3.2 tree ..... 33
  - 2.3.3 Jq ..... 33
- 2.4 本章小结 ..... 34

## 第 3 章 Hyperledger 简介 ..... 35

- 3.1 Hyperledger 综述 ..... 35
  - 3.1.1 Hyperledger 的项目背景 ..... 35
  - 3.1.2 Hyperledger 的项目成员 ..... 36
- 3.2 Hyperledger 的体系结构 ..... 37





3.2.1 获取 Hyperledger 源代码并成为 开发者 .....	37	5.1.1 Fabric 核心模块及其功能 .....	67
3.2.2 Hyperledger 的 9 个正式项目 .....	38	5.1.2 Fabric 模块的通用选项和命令 .....	68
3.3 本章小结 .....	43	5.2 Fabric 模块的子命令、选项和配置 文件 .....	68
<b>第 4 章 Fabric 快速入门</b> .....	44	5.2.1 cryptogen .....	69
4.1 Fabric 的技术特性 .....	44	5.2.2 configtxgen .....	74
4.1.1 Fabric 的多账本特性 .....	44	5.2.3 configtxlator .....	77
4.1.2 Fabric 的智能合约 .....	45	5.2.4 orderer .....	79
4.1.3 Fabric 的权限系统 .....	46	5.2.5 peer .....	85
4.1.4 Fabric 的共识算法 .....	47	5.3 Fabric 模块在系统中的作用 .....	92
4.2 Hyperledger 中与 Fabric 相关的 项目 .....	47	5.3.1 peer 模块在 Fabric 系统中的 作用 .....	92
4.3 Fabric 的模块、安装和使用 .....	48	5.3.2 orderer 模块在 Fabric 系统中的 作用 .....	95
4.3.1 Fabric 的编译和安装 .....	49	5.4 Fabric 数据安全传输的方式 .....	95
4.3.2 Fabric 模块安装结果检查 .....	50	5.4.1 Fabric 中 orderer 模块 TLS 设置 .....	95
4.3.3 利用 Docker 运行 Fabric 相关 模块 .....	51	5.4.2 Fabric 中 peer 模块 TLS 设置 .....	96
4.4 快速运行一个简单的 Fabric 网络 .....	53	5.5 本章小结 .....	98
4.4.1 Fabric 环境准备 .....	53	<b>第 6 章 Fabric 的账号体系</b> .....	99
4.4.2 生成 Fabric 需要的证书文件 .....	54	6.1 Fabric 账号简介 .....	99
4.4.3 创始块的生成 .....	56	6.1.1 Fabric 账号是什么 .....	99
4.4.4 Orderer 节点的启动 .....	59	6.1.2 什么地方需要使用 Fabric 的 账号 .....	101
4.4.5 Peer 节点的启动 .....	60	6.2 基于 cryptogen 的账号管理体系 .....	103
4.4.6 创建通道 .....	64	6.3 Fabric 账号服务器: Fabric-ca .....	106
4.4.7 Chaincode 的部署和调用 .....	65	6.3.1 Fabric-ca 的编译和安装 .....	107
4.5 本章小结 .....	66	6.3.2 fabric-ca-server 的启动和 配置 .....	108
<b>第 5 章 Fabric 核心模块详解</b> .....	67	6.3.3 fabric-ca-client 的使用 .....	115
5.1 Fabric 的核心模块功能、通用 选项和命令 .....	67		

6.4 将 fabric-ca-server 绑定到现有项目中 .....	117
6.5 本章小结 .....	120

## 第 7 章 Fabric 的智能合约详解 .....

7.1 Chaincode 初探 .....	121
7.2 快速编写和运行一个 Chaincode .....	122
7.3 Golang 版本的 Chaincode 的代码结构 .....	125
7.3.1 Chaincode 源代码的基本结构 .....	125
7.3.2 shim 包的核心方法 .....	127
7.3.3 ChaincodeStubInterface 接口中的核心方法 .....	128
7.4 Chaincode 相关的操作命令和选项 .....	136
7.5 如何通过 Chaincode 进行交易的 endorse .....	140
7.6 Chaincode 的调试方法 .....	142
7.6.1 Chaincode 在 Docker 容器之外的运行 .....	142
7.6.2 Chaincode 在 IDE 中的调试 .....	145
7.7 本章小结 .....	148

## 第 8 章 Fabric 和 Fabric-ca 的编程接口 .....

8.1 Fabric 接口的通信协议和功能划分 .....	149
8.2 Fabric Nodejs SDK 的使用 .....	151
8.2.1 如何获取 Fabric Nodejs SDK 源代码 .....	151

8.2.2 快速构建基于 Nodejs 的 Fabric 客户端 .....	151
8.2.3 Fabric Nodejs SDK 中 TLS 的设置 .....	159

## 8.3 Fabric Java SDK .....

8.3.1 Fabric Java SDK 的安装 .....	160
8.3.2 Fabric Java SDK 的常用接口 .....	161
8.3.3 Fabric Java SDK 中 TLS 的设置 .....	169

## 8.4 Fabric Go SDK .....

8.4.1 Fabric Golang 的安装 .....	170
8.4.2 创建配置文件 .....	170
8.4.3 一个简单的 Golang 访问 Fabric 的例子 .....	171
8.4.4 Fabric Golang SDK 其他用法 .....	174
8.4.5 Fabric Golang SDK 的背书操作 .....	176

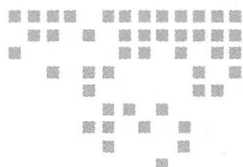
## 8.5 本章小结 .....

## 第 9 章 Fabric 系统架构设计 .....

9.1 Fabric 架构中的组织规划 .....	177
9.1.1 确认组织 .....	178
9.1.2 组织的管理方式 .....	178
9.2 Fabric 系统的结构 .....	179
9.2.1 Fabric 系统的逻辑结构 .....	179
9.2.2 Fabric 系统的物理结构 .....	181
9.3 Fabric 中 Channel 的设计 .....	181
9.4 Chaincode .....	182
9.5 数据访问层 .....	183
9.6 历史遗留系统的兼容 .....	185
9.7 Fabric 系统的维护和管理 .....	186

9.8 本章小结 .....	187	12.1.1 供应链金融的背景知识 .....	230
<b>第 10 章 Fabric 开发实战：开发 流程与实例详解</b> .....	188	12.1.2 供应链金融的痛点 .....	231
10.1 Fabric 项目的开发流程 .....	188	12.1.3 用 Fabric 解决供应链金融 痛点的方法 .....	231
10.2 Fabric 项目开发实例详解 .....	191	12.2 用 Fabric 构建供应链金融系统的 方法 .....	232
10.2.1 系统初始化 .....	191	12.2.1 系统的设计 .....	232
10.2.2 Orderer 节点的初始化和 启动 .....	193	12.2.2 系统环境搭建 .....	233
10.2.3 启动第一个 Peer .....	198	12.2.3 客户端开发 .....	239
10.2.4 Channel 的创建和加入 .....	200	12.3 本章小结 .....	244
10.2.5 启动当前组织的 Fabric-ca .....	202	<b>第 13 章 基于 Fabric 的食品溯源项目 实战</b> .....	245
10.2.6 测试 Chaincode 的部署和 开发 .....	202	13.1 数据溯源的背景知识和痛点 .....	245
10.2.7 客户端的开发 .....	203	13.1.1 数据溯源的背景知识 .....	245
10.2.8 启动本组织的其他 Peer .....	205	13.1.2 数据溯源的痛点 .....	245
10.2.9 其他组织 Peer 节点的加入 .....	208	13.2 Fabric 如何优化数据溯源系统 .....	246
10.2.10 背书交易的测试 .....	210	13.3 Fabric 如何构建数据溯源系统 .....	246
10.2.11 非初始化组织的加入 .....	214	13.3.1 系统环境搭建 .....	247
10.3 本章小结 .....	220	13.3.2 客户端开发 .....	262
<b>第 11 章 基于 Fabric 的区块链浏览器 项目实战</b> .....	221	13.4 本章小结 .....	271
11.1 项目介绍 .....	221	<b>附录 A 比特币的原理和运行方式</b> .....	272
11.2 开发过程 .....	222	<b>附录 B 比特币的 bitcoin-cli 模块 详解</b> .....	282
11.2.1 项目准备 .....	222	<b>附录 C 比特币系统的编程接口</b> .....	292
11.2.2 项目开发 .....	222	<b>附录 D 比特币系统客户端项目 实战</b> .....	297
11.3 本章小结 .....	229	<b>附录 E 区块链相关术语</b> .....	304
<b>第 12 章 基于 Fabric 的供应链金融 项目实战</b> .....	230		
12.1 供应链金融的背景知识和痛点 .....	230		





# 全面认识区块链

人类自诞生以来，一直对物质移动的速度有着孜孜不倦的追求和探索。在人类探索和改造世界的过程中，绝大多数具有颠覆性的技术创新都与物质传递的速度有着非常密切的联系。比如轮子改变人和物体传递的方式，铁轨改变人和物体传递的效率，电力的出现改变了能量的传递方式，互联网的诞生则是彻底颠覆了信息传递的方式和效率。

区块链技术被认为是轮子、铁轨、电力、互联网之后，又一个具备颠覆性的核心技术。作为一种构建价值互联网的底层技术，区块链改变的将是价值传递的方式。区块链的出现将解决人类社会诞生以来一直在思考的问题——如何获取未知的信任。区块链技术到底是怎样一种技术？本章将从宏观角度介绍这个问题。

## 1.1 区块链技术的起源和解释

提到区块链技术，比特币是无法回避的一个重要部分，因为比特币是迄今为止出现最早、规模最大、运行最稳定、技术最成熟的基于区块链技术的应用。2008 年一个网名叫“中本聪”的人发表了一篇名为《比特币：一个点对点的电子现金系统》的论文。在该论文中，“中本聪”描绘了一个完全去中心化的电子现金系统，在这个系统中每一个参与者都是独立并且对等的，这些参与者不依赖于通货保障或者结算交易验证保障的中央权威。

为了实现这套系统，相关的技术社区利用密码学中的椭圆曲线数字签名算法（ECDSA）来实现数据的加密，基于 P2P 网络来实现数据的分布式存储，从而实现了一个去中心化的，不可逆、不可篡改的特殊数据存储系统。这套系统就是目前被称为区块链技术的雏形。比特

币就是构建在区块链技术之上典型的成功应用。比特币系统这些年来稳定而且高效的运行,证明了这些技术理论的正确性和可靠性。

随着业界对比特币系统技术架构的深入了解,人们发现这些技术除了应用在比特币上面之外,还能应用在其他领域。于是相关技术社区将这些技术抽象之后给它们起了一个统一的名字:区块链。从此区块链脱离比特币成为一门单独的技术。

目前区块链已经成为一个独立的技术名词,而不是依赖于某个具体产品的附属技术。区块链这个技术名词,从不同的角度看会有不同的解释。

- 从网络的角度看:区块链的底层网络模型提供了分布式数据存储的完美实现,比特币系统从诞生至今没有发生过一次宕机事件,这有利地证明了该网络模型的稳定和高效。
- 从底层技术的角度看:区块链更像是一个数据结构,用区块存储数据,把区块按照顺序链接起来组成区块链,从而达到防止数据被篡改的目的。
- 从密码学的角度看:区块链利用椭圆曲线数字签名算法来保证数据的完整性和真实性。
- 从数据存储的角度看:区块链更像是一个分布式数据库,不但数据的存储是分布式的(以共享账本为例,所有的数据可以对等地存储在所有参与数据记录的节点中,而非集中存储于中心化的机构节点中),而且数据的产生也是分布式的(账本所有的节点集体维护,而非一个单独的中心机构来维护)。

区块链技术源于比特币但是高于比特币,发展至今,已经形成一个非常完整的技术栈。区块链技术栈中的每个单项技术并不是新发明的技术,如果将这些单项技术单独提取出来,都是比较普通的,但正是这些普通的技术通过精巧地组合之后诞生了一项足以颠覆世界的新技术。这和鸡尾酒非常相识,组成鸡尾酒的每个单独的原料都非常普通,但是组合之后就产生了非常神奇的化学反应,从而诞生了一个让人痴迷的新事物。

套用一句网络流行的话,重要的事情要说三遍:区块链不是一个单独的技术,而是由多种技术组成的技术栈,在学习区块链技术的时候一定要注意区块链技术的这个特性。所以如果想学会区块链技术首先需要对组成区块链技术栈的各个单项技术有所了解,然后再开始学习相关的区块链技术框架,这一点在基于区块链技术的项目实施中尤其重要。

## 1.2 区块链的核心技术及其特性

通过前面章节的介绍我们知道区块链技术是一个技术栈,由多种相关技术组成。那么区块链技术到底是由哪些具体技术组成的呢?在回答这个问题之前,我们先要了解一下区块链具有哪些特点。

### 1.2.1 区块链技术的特性

区块链技术有什么特点？这个问题很多人从不同的角度定义过。在这里，我们引用维基百科上面的说明：“区块链技术是基于去中心化的对等网络，用开源软件把密码学原理、时序数据和共识机制相结合，来保障分布式数据库中各节点的连贯和持续，使信息能即时验证、可追溯，但难以篡改和无法屏蔽，从而创造了一套隐私、高效、安全的共享价值体系。”通过这段描述我们可以把区块链技术的特点归纳为以下几点：

- 区块链没有一个统一的中心，数据分布式存储，并且每个节点是对等的。
- 数据存储按照特定的时序组织并且采用密码学原理加密，这样使得数据不可篡改（密码学加密）并且可以追溯（时序组织）。
- 数据的创建和维护由所有参与方共同参与，任何一方都不能在未经过其他参与方允许的情况下独立对数据进行维护。

这些特性是绝大多数区块链技术的基本特性，但是随着对区块链技术的深入研究，人们发现这些特性已经不能满足业务的需求，因此在区块链技术中增加了一些新的特性，这些新增的特性中最重要的就是智能合约。区块链的智能合约是条款以计算机语言而非法律语言记录的智能合同。智能合约让我们可以通过区块链与真实世界的资产进行交互。当一个预先编好的条件被触发时，智能合约执行相应的合同条款。

通过上面的描述我们可以发现区块链技术具有分布式数据库、密码学、P2P 网络等技术特点。区块链的这些技术特点，使得通过区块链技术可以构建一个去中心化的、安全的、对等的、不可更改的价值传播网络。这些技术通过精巧的组合之后，形成了一种全新的数据记录、传递、存储与展现的方式。以前数据的存储和维护都是由一个统一的中心机构来完成，而区块链技术可以让所有数据的参与方都有机会成为数据维护者。区块链技术在没有中央控制点的分布式对等网络下，使用分布式集体运作的方法，构建了一个 P2P 的自组织网络。通过复杂的校验机制，区块链数据库能够保持完整性、连续性和一致性，即使部分参与者作假也无法改变整个区块链的完整性，更无法篡改区块链中的数据。

现在我们可以对区块链的技术特点进行一下总结。区块链是分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型应用模式。区块链技术栈包含了以下技术特性：

- 分布式数据库的技术特性
- 密码学特性
- 共识机制
- 智能合约

### 1.2.2 区块链的分布式存储技术特性

从技术的特性上看，区块链具有分布式数据库技术的特点。传统的关系性数据库都必



须满足 ACID 原则, ACID 原则本质上是对事务而言的。在传统的关系型数据库中, 事务是一个不能分割的操作单元。因此对于传统的关系数据库而言, 事务必须具备以下四个特性:

- 原子性 (Atomicity): 事务中的所有操作要么全部执行, 要么全部拒绝, 没有任何中间状态;
- 一致性 (Consistency): 数据库的完整性约束不会被任何事务破坏;
- 隔离性 (Isolation): 多个事务完全隔离开来, 一个事务的执行不会被其他事务所影响;
- 持久性 (Durability): 一个事务完成之后, 该事务对数据库的变更会被永久地存在数据库中。

从 ACID 四个属性我们看出, 区块链可以满足上面的部分特性。

- 原子性, 区块链的数据存储在区块中, 一个区块链中的数据要么全部进入区块链, 要么全部被丢弃。
- 一致性, 区块加入区块链之后原有的区块链保持不变。
- 隔离性, 所有节点可以同时生成区块, 但是最终只有一个区块可以加入区块链中。
- 持久性, 一旦区块加入区块链中, 就会被永久保存并复制到其他节点。

移动互联网对数据的存储数量以及数据的读写速度都提出了更高的要求, 因此基于 ACID 的关系数据已经不能满足于业务的需求。相关技术社区在原有的 ACID 数据库的基础上创建了分布式数据库系统。分布式数据库系统有这样一些特点, 我们称之为 BASE。BASE 是一组单词的首字母缩写, 它们是:

- 基本上可用 (basically available): 主要的需求是可用性, 即使出现划分的情况下, 也应该允许更新, 哪怕以牺牲一致性为代价;
- 软状态 (soft state): 网络划分可能导致数据库每个副本都有一定程度不同的状态, 从而导致整体状态不明;
- 最终一致性 (eventually consistent): 当解决完划分后, 要求最终所有副本形成一致。

和 ACID 的强一致性概念比较, BASE 面向的是可扩展的分布式系统。BASE 在牺牲强一致性的基础上换取了可用性, 允许在某个时间段内不同节点之间存在数据的不一致性, 但是最终所有节点的数据都是一致的。而区块链的节点是分布在全世界各个地方的, 在一定的时段内, 不同节点的区块数存在不一致的情况, 但是最终都是一致的。所以我们认为区块链是符合分布式数据库 BASE 规则的。通过上面的对比我们发现, 区块链符合传统的关系数据库和互联网时代的分布式数据库特性。

### 1.2.3 区块链的密码学技术特性

为了保证数据的不可逆、不可篡改和可追溯, 区块链采用了一些密码学相关的技术。主要使用的是哈希算法、Merkle 树、非对称加密算法这三种密码学中常用的技术。

## 1. 哈希算法

哈希算法将任意长度的二进制值映射为较短的固定长度的二进制值，这个小的二进制值称为哈希值。哈希值是一段数据唯一且极其紧凑的数值表示形式。如果哈希一段明文而且哪怕只更改该段落的一个字母，随后的哈希值都会发生变化。要找到哈希值相同而输入值不同的字符串，在计算上是不可能的。所以数据的哈希值可以检验数据的完整性。在哈希算法中如果输入数据有变化，则哈希也会发生变化。哈希算法可用于许多操作，包括身份验证和数字签名（也称为“消息摘要”），不过一般用于快速查找和加密算法。

区块链的数据是存储在区块中的，每个区块都有一个区块头，区块头存储区块中所有数据经过哈希算法获取的一个哈希值，同时每个区块中存储前面一个区块的哈希值，这样每个区块都会通过所存储的前一个区块的哈希值串联起来，这样就形成了区块链。如果有人试图篡改其中的一笔交易，势必会导致该交易所在区块的哈希值发生变化，为了使得被篡改的交易得到所有节点的认可，篡改者需要以被篡改的节点为起点，重新计算后面的所有区块，但是如果要让所有的节点都承认和接受这些篡改，那基本上是不可能完成的事情了。从这里我们可以发现哈希算法的应用使得篡改的成本已经远远超过收益了。

区块链系统常用的哈希算法是 SHA256 和 RIPEMD160。SHA256 是 SHA 算法的一个变体。SHA（安全散列算法）是由美国国家安全局（NSA）设计，美国国家标准与技术研究院（NIST）发布的一系列密码散列函数，包括 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512 等变体。这些算法主要适用于数字签名标准（Digital Signature Standard DSS）里面定义的数字签名算法（Digital Signature Algorithm DSA）。SHA256 算法在抗碰撞性和效率之间做了一个平衡处理，在很多区块链系统中均支持 SHA256 哈希算法。

---

对于哈希算法来说，抗碰撞性越高，相对需要的计算资源越大，对系统性能也会有一定的影响。因此很多区块链系统可以通过配置参数修改算法，使用者可以根据业务需求选择合适的哈希算法。

---

## 2. Merkle 树

通过前面的描述我们知道区块中的数据是存储在区块中的，一个区块中会存储若干数据，那么这些数据是以什么样的方式组织才能够做到不可篡改呢？Merkle 树解决了这个问题。

### （1）什么是 Merkle 树

Merkle 树是一种树（数据结构中所说的树），通常称为 Merkle Hash Tree。组成 Merkle 树的所有节点都是哈希值。Merkle 树具有以下特点：

- Merkle 树是一种树型数据结构，可以是二叉树也可以是多叉树，具有树型结构的所有特点；



- Merkle 树的叶子节点上的 value 可以任意指定, 比如可以将数据的哈希值作为叶子节点的值;
- 非叶子节点的 value 是根据它下面所有的叶子节点值, 然后按照一定的算法计算得出的。如 Merkle 树的非叶子节点 value 是将该节点的所有子节点进行组合, 然后对组合结果进行哈希计算所得出的哈希值。

## (2) Merkle 树的应用领域

目前, 在计算机领域 Merkle 树多用来进行比对以及验证处理。比特币钱包服务用 Merkle 树的机制来做“百分百准备金证明”。在处理比对或验证的应用场景中, 特别是在分布式环境下进行比对或验证时, Merkle 树可以大大减少数据的传输量以及计算的复杂度。

## (3) Merkle 树的优点

Merkle 树明显的一个好处是可以单独拿出一个分支(作为一个小树)来对部分数据进行校验, 这个特性在很多使用场合可以带来哈希列表所不能比拟的方便和高效。

## (4) Merkle 树在区块链中的应用

在区块链中, 区块中的交易是按照 Merkle 的形式存储在区块上面的。每笔交易都有一个哈希值, 然后不同的哈希值向上继续做哈希运算, 最终形成了唯一的 Merkle 根。这个 Merkle 根将会被存放到区块的区块头中。利用 Merkle 树的特性可以确保每一笔交易都不可伪造。

## 3. 非对称加密算法

加密算法一般分为对称加密和非对称加密, 非对称加密是指为满足安全性需求和所有权验证需求而集成到区块链中的加密技术。非对称加密通常在加密和解密过程中使用两个非对称的密码, 分别称为公钥和私钥。非对称密钥对具有两个特点: 一是用其中一个密钥(公钥或私钥)加密信息后, 只有另一个对应的密钥才能解开; 二是公钥可向其他人公开, 私钥则保密, 其他人无法通过该公钥推算出相应的私钥。

非对称加密一般划分为三类主要方式: 大整数分解问题类、离散对数问题类、椭圆曲线类。大整数分解问题类指用两个较大的质数的乘积作为加密数, 由于质数的出现具有不规则性, 想要破解只能通过不断试算。离散对数问题类指的是基于离散对数的难解性, 利用强的单向散列函数的一种非对称分布式加密算法。椭圆曲线类指利用平面椭圆曲线来计算成组非对称特殊值, 比特币就使用此类加密算法。

非对称加密技术在区块链的应用场景主要包括信息加密、数字签名和登录认证等。其中信息加密场景主要是由信息发送者(记为 A)使用接受者(记为 B)的公钥对信息加密后再发送给 B, B 利用自己的私钥对信息解密。比特币交易的加密即属于此场景。数字签名场景则是由发送者 A 采用自己的私钥加密信息后发送给 B, 再由 B 使用 A 的公钥对信息解密,



从而可确保信息是由 A 发送的。登录认证场景则是由客户端使用私钥加密登录信息后发送给服务器,后者接收后采用该客户端的公钥解密并认证登录信息。

存储在区块链上的交易信息是公开的,但是账户身份信息是高度加密的,只有在数据拥有者授权的情况下才能访问,从而保证了数据的安全和个人的隐私。区块链系统每个用户都有一对秘钥,一个是公开的,一个是私有的。通常公钥的密码算法采用的是椭圆曲线算法。用户可以通过自己的私钥对交易进行签名,同时别的用户可以利用签名用户的公钥对签名进行验证。

椭圆曲线指的是由韦尔斯特拉斯(Weierstrass)方程  $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$  所确定的平面曲线。若  $F$  是一个域,  $a_i \in F, i=1, 2, \dots, 6$ 。满足式(1-1)的数偶  $(x, y)$  称为  $F$  域上的椭圆曲线  $E$  的点。 $F$  域可以是有理数域,还可以是有限域  $GF(Pr)$ 。椭圆曲线通常用  $E$  表示。除了曲线  $E$  的所有点外,尚需加上一个叫作无穷远点的特殊点  $O$ 。在椭圆曲线加密(ECC)中,利用了某种特殊形式的椭圆曲线,即定义在有限域上的椭圆曲线。其方程如下:

$$y^2 = x^3 + ax + b(\text{mod } p) \quad (1-1)$$

这里  $p$  是素数,  $a$  和  $b$  为两个小于  $p$  的非负整数,它们满足:

$$4a^3 + 27b^2(\text{mod } p) \neq 0 \quad (1-2)$$

其中,  $x, y, a, b \in Fp$ , 则满足式(1-2)的点  $(x, y)$  和一个无穷点  $O$  就组成了椭圆曲线  $E$ 。

椭圆曲线离散对数问题 ECDLP 定义如下:给定素数  $p$  和椭圆曲线  $E$ , 对  $Q = kP$ , 在已知  $P, Q$  的情况下求出小于  $p$  的正整数  $k$ 。可以证明,已知  $k$  和  $P$  计算  $Q$  比较容易,而由  $Q$  和  $P$  计算  $k$  则比较困难,至今没有有效的方法来解决这个问题,这就是椭圆曲线加密算法原理之所在。与 RSA 算法相比,椭圆曲线公钥系统是代替 RSA 的强有力的竞争者。椭圆曲线加密方法的优点总结如下:

- 安全性能更高,如 160 位 ECC 与 1024 位 RSA、DSA 有相同的安全强度。
- 计算量小,处理速度快,在私钥的处理速度上(解密和签名),ECC 远比 RSA、DSA 快得多。
- 存储空间占用小,ECC 的密钥尺寸和系统参数与 RSA、DSA 相比要小得多,所以占用的存储空间小得多。
- 带宽要求低,使得 ECC 具有广泛的应用前景。

ECC 的这些特点使它必将取代 RSA,成为通用的公钥加密算法。比如 SET 协议的制定者已把它作为下一代 SET 协议中缺省的公钥密码算法。

利用椭圆曲线的签名和验证算法,可以保证账号的唯一性和不可冒名顶替性,同时也保证了用户的不可抵赖性。通过这些密码学技术的应用可以使得区块链技术在没有中心服务器的情况下做到数据的不可逆和不可篡改。

### 1.2.4 区块链中的共识机制

区块链中的一个核心概念是去中心，在区块链中没有和传统数据库系统一样的中心数据库，每个节点都是对等的，这样就需要一套算法和机制来保证所有对等节点之间可以有效协作。这套算法和方式称为共识机制。共识机制的存在可以有效保证各个节点之间按照既定的原则共同维护账本。共识机制本质上是区块链系统中实现不同节点之间建立信任、获取权益的数学算法。共识算法在区块链技术出现之前就已经存在，最早出现在分布式系统中。目前区块链系统中常用的共识算法有 PWO（工作量证明）、POS（股权证明机制）、DPOS（授权股权证明）、PBFT（拜占庭共识算法）。

#### 1. POW——工作量证明机制

工作量证明（Proof Of Work, POW），简单理解就是一份证明，用来确认你做过一定量的工作。监测工作的整个过程通常是极为低效的，而通过对工作的结果进行认证来证明完成了相应的工作量，则是一种非常高效的方式。比如现实生活中的毕业证、驾驶证等，也是通过检验结果的方式（通过相关的考试）所取得的证明。

工作量证明系统主要特征是客户端需要做一定难度的工作得出一个结果，验证方却很容易通过结果来检查出客户端是不是做了相应的工作。这种方案的一个核心特征是不对称性：工作对于请求方是适中的，对于验证方则是易于验证的。它与验证码不同，验证码的设计出发点是易于被人类解决而不易被计算机解决。

举个例子，给定一个基本的字符串“study blockchain!”，我们给出的工作量要求是，可以在这个字符串后面添加一个整数值，对变更后的字符串进行 SHA256 哈希运算，如果得到的哈希结果（以十六进制的形式表示）是以“0000”开头的，则验证通过。为了达到这个工作量证明的目标。我们需要不停地递增这个整数的值，对得到的新字符串进行 SHA256 哈希运算。按照这个规则，我们需要经过 79505 次计算才能找到恰好前 4 位为 0 的哈希散列。

```
"study blockchain!0" => 199dd3e519f1e170e0eea67744547ffea5a435e4ddacaf0ada859b
972091ac7d "study blockchain!1" => 91de0a6ce666ee00ee8cb5893cc843b84001a4c9ffac5bc
3168b3c6c6f604217 "study blockchain!2" => 86d4fc2522d934af711049e4bd488a50a86bf2d6
57f1d9e1871c9f25d4f6ea2b ... "study blockchain!79505" => 000024f2c9e0e1cf93ef68fde
a6c60e4dd0498d41aa82473c41c2f1b022b4ca1
```

通过这个示例我们对工作量证明机制有了一个初步的理解。有的人会认为如果工作量证明只是这样的一个过程，那是不是只需要记住字符串后面的数字为 79505 计算能通过验证就行了？当然不是的，这是一个非常简单的例子，实际会增加很多的参数来调节计算的难度。

比特币系统的共识机制就是基于 POW 算法的，但是远远比上面的例子要复杂得多。比特币网络中任何一个节点如果想生成一个新的区块并写入区块链中，必须解出比特币网络给



出的工作量证明的谜题。这道题关键的三个要素是工作量证明函数、区块及难度值。工作量证明函数是这道题的计算方法，区块决定了这道题的输入数据，难度值决定了这道题所需要的计算量。

## 2. POS——股权证明机制

股权证明机制的基本概念是产生区块的难度应该与你在网络里所占的股权（所有权占比）成比例。简单来说 POS 就是一个根据你持有货币的量和时间给你发利息的一个制度。在 POS 模式下有一个名词叫币龄，每个币每天产生 1 币龄。比如你持有 100 个币，总共持有了 30 天，那么此时你的币龄就为 3000。这个时候如果你发现了一个 POS 区块，你的币龄就会被清空为 0。你每被清空 365 币龄，你将会从区块中获得 0.05 个币的利息（可理解为年利率 5%）。那么在这个案例中， $\text{利息} = 3000 * 5\% / 365 = 0.41$  个币，这样只要持有货币就可以获取利息（需要注意的是，5% 的年利率仅仅是我们举例，并非每个 POS 模式的币种都是 5%）。

POS 的设计理念以及初衷主要是基于以下三个原因：

第一：众所周知，比特币的区块产量每 4 年会减半，未来随着比特币区块包含产量的逐步降低，挖矿的动力将会不断下降，矿工人数越来越少。整个比特币网络有可能会逐渐陷入瘫痪（因为大家都减少了运行比特币客户端的时间，越来越难找到一个 P2P 节点去连接和同步网络数据）。针对这个问题，POS 提供了解决方案：在 POS 体系中只有打开钱包客户端程序才能发现 POS 区块，才会获得利息。这促使很多不想挖矿的人也会常常打开自己的钱包客户端，通过这样的方法使得整个网络更加健壮。

第二：通过比特币的原理我们知道在若干年后随着矿工人数的下降，比特币很有可能被一些高算力的人、团队或者矿池所挟持，进而进行 51% 攻击。这会导致整个比特币网络崩溃。51% 攻击简单来说就是当你拥有了超过全球 51% 的比特币算力时，你将能伪造比特币网络的任何数据。比如你伪造自己有一百万个比特币（实际上你没有），你可以通过你的算力强迫其他节点接受你的虚假交易，但是这样将导致整个体系的崩溃。针对这个问题，POS 的解决方案是：在 POS 体系中即使你拥有了全球 51% 的算力也未必能够进行 51% 攻击。因为有一部分的货币并不是挖矿产生的，而是由利息产生（利息存放在 POS 区块中），这要求攻击者还需要持有全球超过 51% 的货币量，这将大大提高 51% 攻击的难度。

第三：虽然我们知道比特币是一个永远不会通货膨胀的体系，因为它的货币总量表面看起来是固定的，但是我们应该知道比特币其实是一个通货紧缩的体系。因为当我们重装了系统或者忘记了钱包密钥的时候，我们会永远无法再拿回钱包里的钱。这意味着每年都会有一些比特币随着钱包的丢失而永远被锁定，这就形成了实质上的通货紧缩。也许在五十年后，有效的比特币将会只剩下一千万个或者更少。POS 部分解决了这个问题，在 POS 体系中由



于一部分货币是由利息产生的, 因此整个体系中会不断地产生新的货币。

### 3. DPOS——委托权益证明

在区块链中共识算法被用来保证整个区块链网络的安全可靠。在前面我们介绍了工作量证明 (POW) 和权益证明 (POS) 这两种共识算法。但是这两种共识算法都不能解决交易性能问题, 尤其是 POW 算法大量消耗计算所需的电力。而委任权益证明 (DPOS) 共识算法正是为了解决这些问题而诞生的。

委任权益证明 (Delegated Proof of Stake, DPOS) 最初由比特股 (BitShares) 提出并采用。目前 DPOS 已经在除了比特股之外的多个区块链技术平台上面可靠运行。这足以证明 DPOS 算法是健壮、安全和有效的。在 DPOS 算法中持有一定数据量货币的可成为股东, 每个股东按其持股比例拥有影响力。超过 51% 股东投票的结果将是不可逆且有约束力的。为达到这个目标, 每个股东可以将其投票权授予一名代表。获票数最多的前 100 位代表按既定时间表轮流产生区块。每名代表分配到一个时间段来生产区块。所有的代表将收到等同于一个平均水平的区块所含交易费的 1% 作为报酬。如果一个平均水平的区块含有 100 股作为交易费, 一名代表将获得 1 股作为报酬。网络延迟有可能使某些代表没能及时广播他们的区块, 理论上将导致区块链分叉。但是这种现象在 DPOS 算法中不太可能发生, 因为制造区块的代表可以与制造前后区块的代表建立直接的联系。建立这种与你之后的代表 (也许也包括其后的那名代表) 的直联系是为了确保你能得到报酬。这种模式可以每 30 秒产生一个新区块, 并且在正常的网络条件下区块链分叉的可能性极其小, 即使发生也可以在几分钟内得到解决。由此可见 DPOS 算法是对 POS 算法的有效补充。

### 4. PBFT——拜占庭共识算法

PBFT 算法是根据拜占庭问题演变而来的拜占庭共识算法。在拜占庭问题被提出后一直有各种共识算法来解决拜占庭问题, 但是无论从执行流程的复杂度还是算法效率来说, PBFT 是目前公认效率最好的算法。该算法是 Miguel Castro (卡斯特罗) 和 Barbara Liskov (利斯科夫) 在 1999 年提出来的。PBFT 算法有效地解决了原始拜占庭容错算法效率不高的问题, 将算法复杂度由指数级降低到多项式级, 使得拜占庭容错算法在实际系统应用中变得可行。

关于拜占庭将军问题, 一个简易的非正式描述如下:

拜占庭帝国想要进攻一个强大的敌人, 为此派出了 10 支军队去包围这个敌人。这个敌人虽不比拜占庭帝国, 但也足以抵御 5 支常规拜占庭军队的同时袭击。基于一些原因, 这 10 支军队不能集合在一起单点突破, 必须在分开的包围状态下同时攻击。他们任一支军队单独进攻都毫无胜算, 除非有至少 6 支军队同时袭击才能攻下敌国。他们分散在敌国的四周, 依靠通信兵相互通信来协商进攻意向及

进攻时间。困扰这些将军的问题是，他们不确定队伍中是否有叛徒，叛徒可能擅自变更进攻意向或者进攻时间。在这种状态下，拜占庭将军们能否找到一种分布式的协议来让他们能够远程协商，从而赢得战斗？这就是著名的拜占庭将军问题。

应该明确的是，拜占庭将军问题中并不去考虑通信兵是否会被截获或无法传达信息等问题，即消息传递的信道绝对可靠。Lamport 已经证明了在消息可能丢失的不可靠信道上试图通过消息传递的方式达到一致性是不可能的。所以，在研究拜占庭将军问题的时候，我们已经假定了信道是没有问题的，并在这个前提下，去做一致性和容错性相关研究。

### 5. Casper——投注共识

Casper 是以太坊提出的下一代的共识机制，从原理上看，Casper 属于 POS。Casper 的共识是按块达成的，而不是像 POS 那样按链达成。为了防止验证人在不同的世界中提供不同的投注，这里还有一个简单严格的条款：如果你有两次投注序号一样，或者说你提交了一个无法让 Casper 合约处理的投注，你将失去所有保证金。从这一点我们可以看出，与传统的 POS 不同，Casper 有惩罚机制，这样非法节点通过恶意攻击网络不仅得不到交易费，还面临着保证金被没收的风险。

### 6. Ripple Consensus——瑞波共识算法

瑞波共识算法使一组节点能够基于特殊节点列表达成共识。初始特殊节点列表就像一个俱乐部，要接纳一个新成员必须由该俱乐部 51% 的会员投票通过。共识遵循这核心成员的 51% 权力，外部人员则没有影响力。由于该俱乐部由“中心化”开始，它将一直是“中心化的”，如果它开始腐化，股东们什么也做不了。瑞波系统将股东们与其投票权隔开，并因此比其他系统更中心化。

### 7. POET——消逝时间量证明

POET (Proof of Elapsed Time) 共识算法的大致思路是这样的，每个节点发布块之前都要从一个 enclave (在 Sawtooth 中它代表一个可信操作) 获取一个随机的等待时间，等待时间最短的率先发布块 (相当于被选为 leader)，其中 enclave 是通过新型的安全 CPU 指令来实现的。enclave 支持两个函数 “CreateTimer” 和 “CheckTimer”，CreateTimer 用于从 enclave 中产生一个 timer，CheckTimer 会去校验这个 timer 是不是由 enclave 产生、是否已经过期。如果满足以上两个条件就会生成一个 attestation (凭证)。attestation 中包含的信息可以用来校验 certificate 是否由该 enclave 产生并且已经等待了 timer 规定的时间。成为 leader 的概率与捐献的资源是成比例的，因为是通用处理器而不需要定制矿机，所以参与的门槛比较低，节点会比较多，整个共识会更健壮。

在前面的内容中我们简单地介绍了区块链常用的共识算法。不同的区块链平台根据自



身的技术特点采用了不同的共识算法。但是有一点需要说明一下, 这些共识算法是区块链的底层核心, 在一个已经实现的区块链技术平台中, 共识算法相关的模块都已经实现, 在实际项目开发不会需要架构师或者开发人员去实现这些算法, 但是了解这些算法将有助于我们更好地设计基于区块链的系统架构。

### 1.2.5 区块链中的智能合约

提到智能合约首先需要说明的是, 智能合约和区块链原本是两个独立的技术。在区块链诞生之初并没有引入智能合约的概念。比如, 在以比特币为代表的区块链 1.0 系统中并没有智能合约的概念。随着区块链技术的发展, 人们发现区块链在价值传递的过程中需要有一套规则来描述价值传递的方式, 这套规则应该让机器来识别和执行而不是人, 因此智能合约进入了人们的视线之内。以太坊的出现让这种假设成为可能, 而实现的方式正是依靠智能合约。

智能合约的理念可以追溯到 1995 年, 几乎与互联网同时出现。密码学家尼克·萨博 (Nick Szabo) 首次提出了“智能合约”这一术语。从本质上讲, 这些自动合约的工作原理类似于其他计算机程序的 if-then 语句。智能合约只是以这种方式与真实世界的资产进行交互。当一个预先编好的条件被触发时, 智能合约执行相应的合同条款。

但是在尼克·萨博提出智能合约的工作理论后, 智能合约相关的技术迟迟无法落地, 而且很长时间也没有相关的产品问世。产生这种现象的一个重要原因是因为缺乏能够支持可编程合约的数字货币系统和技术。区块链技术的出现解决了该问题, 它不仅可以支持可编程合约, 而且具有去中心化、不可篡改、过程透明可追踪等特性, 这些特性天然适用于智能合约。目前智能合约技术已经成为区块链技术的必备特性之一。

在区块链 2.0 引入智能合约之后, 区块链真正地脱离了数字货币的枷锁, 成为一个独立的技术。因为智能合约的引入, 区块链可以应用在更加广泛的场景中。智能合约可以认为是区块链技术的翅膀, 让区块链飞得更高更远。既然智能合约对区块链如此重要, 那么智能合约到底是什么呢?

智能合约本质上就是一段用某种计算编程语言编写的程序, 这段程序可以运行在区块链系统提供的容器中, 同时这段程序也可以在某种外在、内在条件的激活下自动运行。这样的特性和区块链技术结合之后不但可以避免人为对规则的恶意篡改, 而且可以发挥智能合约在效率和成本方面的优势。由于智能合约的代码是存放在区块链中, 智能合约的运行也是在区块链系统提供的容器之中的, 结合区块链技术所使用的密码学原理, 使得智能合约天然具有防篡改和防伪造的特性。智能合约产生的结果也是存储在区块中的, 这样从源头、执行过程到结果全程都在区块链中执行, 保证了智能合约的发布、执行、结果记录的真实性和唯一性。



智能合约技术首先在以太坊得以实现，在超级账本的 Fabric 项目也引入了智能合约的概念。在后续章节中将详细介绍如何在这些平台中通过智能合约开发应用。

## 1.3 区块链技术演进过程

通过上面的介绍我们可以发现区块链技术为了使用新的业务场景而一直在改进中。到目前为止区块链技术已经进入 3.0 时代。

- 区块链 1.0 仅仅是一个共享账本，只能记账而没有其他功能。
- 区块链 2.0 在共享账本的基础上增加了智能合约，通过智能合约可以提供更加丰富的功能。
- 区块链 3.0 进一步升级，不但能够记录交易还能记录状态，对数据进行溯源，使区块链技术不再局限于数字货币，而是应用在更多的行业场景中。

区块链的三个版本之间并没有取代关系，每个版本都有自己的特点，它们之间没有必然的关系并且是相互独立的。到目前为止，每个版本的区块链技术依然在各自擅长的领域发挥着重要的作用，有很多经典的应用在运行。本书将挑选其中比较有代表性的比特币和以太坊作为代表，为大家介绍这两个平台的技术特性和使用方法。

## 1.4 区块链技术的 3 个缺点

区块链技术虽然有美好的未来和前景，但是作为一项新兴的技术，和其他技术一样，会有一个完善的过程。目前的区块链系统还有一些不足和需要改进的地方，主要有以下几个问题。

### 1. 性能问题

区块链由于其在数据完整性和不可篡改性等方面的特殊要求，每笔交易均需要打包到区块中，然后通过计算每笔交易的 Hash 值，从而构造一个完整的 Merkle 树，最终将交易保存到区块中。这样的处理方式保证了数据的安全性和完整性，但是速度会大幅下降。以比特币系统为例，目前比特币系统每秒只能处理大约 10 笔交易，这显然是不能满足实际的业务需求的。虽然针对性能问题也提出了很多解决方案，比如闪电网络、石墨烯等技术，但是这些技术方案大多数还处于技术验证中，距离实际的应用还有一段距离。

### 2. 数据的弹性扩展问题

区块链系统具有分布式系统的特性，但是到目前为止，区块链系统只能做到节点的分布式，在数据存储上还没有提供可靠的分布式解决方案。比如比特币的所有交易数据已经多

达 150G 左右, 并且只能部署在单台机器上。随着时间的推移, 这些交易数据只增不减, 为了应付不断增长的交易数据, 只能不断增加单台主机的存储。这种存储方式在遇到存在海量数据的业务场景中会带来隐患。

### 3. 易用性问题

区块链技术是新兴技术, 虽然单个技术已经出现很久, 但是这些技术组合之后产生了很多新的特性。目前技术社区普遍还处于早期阶段, 相关的案例、技术文档、技术社区等普遍比较缺失。这些因素导致了区块链技术在学习、推广、落地方面出现了不同程度的障碍。这些障碍的解决还需要整个技术社区继续努力。

## 1.5 区块链技术常见的 4 个错误认识

### 1. 区块链就是数字货币

区块链技术源于比特币, 可以说比特币是一个成功的区块链应用。不仅仅是比特币, 目前的以太坊等数字货币都是基于区块链理论而实现的区块链产品, 但是区块链和数字货币之间不能画上等号。区块链是一个由多技术组成的技术栈, 而数字货币是基于区块链技术的一个产品, 区块链除了在数字货币领域之外在很多其他的领域都有应用。

### 2. 区块链将取代传统的数据库

区块链具有分布式数据库的特性, 但是区块链绝不是为了取代传统的数据库系统。它们解决的问题是不一样的。在未来二者之间更多的是合作的关系。

### 3. 区块链系统是否一定要挖矿

挖矿是 POW 共识算法中的一个行为, 是比特币等数字货币对 POW 算法的一种实现方式, 但是挖矿绝对不是区块链的必需品。不是所有的区块链技术平台都需要挖矿。比如 Fabric 就没有采用 POW 的共识机制, 也就没有挖矿这一说了。目前的主流平台中以太坊在将来也可能会支持 POS 的共识算法。

### 4. 区块链只能用来记账

区块链技术源于比特币, 而比特币是一个数字货币系统, 所有的记录都是和交易相关的, 因此自然就把这些数据集称为账本。在比特币之后的区块链系统中习惯把区块链中存数据的集合称为账本, 把每一条数据集称为交易。包括很多区块链系统的源码中涉及数据存储和单条数据的变量命名中都包括 Ledger 和 Transaction 等单词。这样给人的感觉是区块链是用来记账的。这其实是个误会, 区块链技术发展到今天, 其应用范围远远超出了数字货币的范畴, 在很多领域均有广泛的应用。而且区块链中存储的数据可以是任何数据, 甚至包括图片和视频。

## 1.6 区块链技术的应用领域

随着区块链技术的成熟和普及,很多行业都在积极探索利用区块链技术来解决本行业的痛点问题。目前区块链技术主要在以下几个行业得到了相对广泛的应用。

### 1.6.1 区块链在金融行业的应用

区块链源于比特币,而比特币本身具有货币属性,因此区块链和金融服务有天然的结合点。不仅如此,由于区块链技术所拥有的高可靠性、简化流程、交易可追踪、节约成本、减少错误以及改善数据质量等特质,使得其具备重构金融业基础架构的潜力。

#### 1. 金融行业痛点

目前金融行业最大的问题是每个机构都有自己的账本,机构之间在进行业务对接的时候需要进行大量的对账、清算、结算等操作,这些操作都需要耗费大量的人力和物力。这不仅导致了用户端和金融机构中后台业务端等产生的支付业务费用高昂,也使得小额支付业务难以开展。在票据及供应链金融领域,业务因人为介入多,存在许多违规事件及操作风险。票据业务创造了大量流动性的同时,相关市场也容易滋生出违规操作或用户欺诈行为,进而导致商业银行的汇票业务事件集中爆发。

国内现行的汇票业务仍有约70%为纸质交易,操作环节处处需要人工,并且因为涉及较多中介参与存在管控漏洞、违规交易的违规操作从而提高了风险。同时由于机构之间的信息不对称很容易导致凭证伪造事件的发生。在证券领域,证券交易生命周期内的一系列流程耗时较长,增加了金融机构中后台的业务成本。在清算和结算领域,由于不同金融机构间的基础设施架构、业务流程各不相同,同时在具体操作过程中涉及很多人工处理的环节,这些因素极大地增加了业务成本同时也容易出现差错。在用户身份识别领域,不同金融机构间的用户数据难以实现高效的交互,使得重复认证成本较高,间接带来了用户身份被某些中介机构泄露的风险。

#### 2. 区块链在金融行业的作用

区块链技术通过密码学原理从底层解决了数据的不可篡改和可追溯特性,利用这些特性监管机构可以非常方便地对整个交易过程实施精准、及时的监管。在事后追责的时候可以快速精准地获取证据。同时由于区块链是一个对等的共享账本,价值可以直接通过区块链进行安全高效的转移,这些特性可以节省大量清算以及结算相关的费用并且简化流程。

#### 3. 区块链在金融行业的应用场景

##### 应用场景1:支付

在支付领域区块链可以摒弃中转银行的角色,实现点对点快速且成本低廉的跨境支付。



通过区块链的平台不但可以绕过中转银行以减少中转费用,还因为区块链安全、透明、低风险的特性提高了跨境汇款的安全性。同时由于区块链去中心化的特性可以加快结算与清算速度以提高资金利用率。在未来银行与银行之间可以不再通过第三方而是通过区块链技术以点对点方式进行支付。由于省去第三方金融机构的中间环节,不但可以实现全天候支付、实时到账,而且有助于降低跨境电商资金风险及满足跨境电商对清算服务及时性、便捷性的需求。

#### 应用场景 2: 票据与供应链金融业务

区块链可以实现票据价值传递的去中介化。长久以来票据的交易一直存在一个第三方的角色来确保有价值凭证的传递是安全可靠的。在纸质票据中交易双方的信任建立在票据真实性的基础上,即使在现有电子票据交易中,也是需要通过央行 ECDS 系统的信息进行交互认证。但借助区块链的技术可以直接实现点对点之间的价值传递,不需要特定的实物票据或中心系统进行控制和验证。在供应链金融中也能通过区块链减少人工成本、提高安全度以及实现端到端的透明化。未来通过区块链技术供应链金融业务将能大幅减少人工的介入。所有参与方(包括供货商、进货商、银行)都能使用一个去中心化的账本共享文件并在达到预定的时间和结果时通过智能合约自动进行支付,这样将极大地提高效率并且能有效地减少交易中由于人工误操作而造成的损失。

### 1.6.2 区块链在供应链中的应用

产品从生产到销售,从原材料到成品直至最后抵达客户手里这个过程中涉及的所有环节都属于供应链的范畴。在供应链领域中多流程、多参与方的特性给区块链切入供应链系统提供了天然的基础。

#### 1. 供应链行业的痛点

目前供应链系统中可能涉及几百个加工环节,几十个不同的地点,数目庞大的节点给供应链的追踪管理带来了很大的困难。由于供应链的整个过程并不公开,因此消费者没有办法确切核实所购买产品的真正价值,这也就意味着产品的价格是否与其价值相符我们也无从得知。此外供应链过程中若出现了非法经营活动时,执法人员该从哪里开始调查、该向谁问责一切都不容易。这些弊端直接导致了市面上仿冒产品泛滥,工厂老板强迫工人超负荷工作,工厂卫生条件不合格等违规现象的发生。

#### 2. 区块链在物流行业中的应用

区块链作为一种分布式总账系统能够提高行业的透明度和安全性。相关专家看好区块链技术,相信它能够解决供应链目前存在的问题。区块链技术可以在不同分类账上记录下产品在供应链过程中涉及的所有信息,这些信息包括负责企业、价格、日期、地址、质量,以及产品状态等有用信息。由于共享账本具有可查询的特点,因此当客户想了解产品的时候只

要查找相关分类账,这样无论是产品的原材料、产品整个加工工艺等信息都可以搜索到。因为分布式账本的分散性特点,相关厂家想要篡改产品数据几乎是不可能的。由于交易过程中所有的数据都是加密保护的,所以数据被窃取的问题是可以避免的。基于上述特性,我们认为区块链技术是改进供应链问题的最佳技术之一。

### 3. 区块链在物流行业的应用场景

#### 应用场景 1: 物流

在物流过程中利用数字签名和公私钥加解密机制可以充分保证信息安全以及寄、收件人的隐私。例如,快递交接需要双方私钥签名,每个快递员或快递点都有自己的私钥,是否签收或交付只需要查一下区块链即可。最终用户没有收到快递就不会有签收记录,快递员也无法伪造签名,因此通过区块链可杜绝快递员通过伪造签名来逃避考核的行为,在减少用户投诉的同时还能有效地防止货物的冒领和误领。由于区块链的匿名性,真正的收件人并不需要在快递单上直观展示实名制信息,因此个人信息得到了保障。通过区块链技术的安全性,更多人会愿意接受实名制,从而促进国家物流实名制的落实。最后利用区块链技术中的智能合约功能,可以有效地简化物流程序并且大幅度提升物流的效率。

#### 应用场景 2: 溯源防伪

区块链技术也可用于药品、艺术品、收藏品、奢侈品等的溯源防伪。我们以钻石为例,可以在钻石身份认证及流转过程中为每一颗钻石建立唯一的电子身份。同时记录每一颗钻石的属性并存放至区块链中。这样这颗钻石的来源、流转历史记录、归属或者所在地会被记录在链中,而且区块链中的数据天然具有防篡改性。通过区块链记录钻石的属性信息,在遇到诸如非法的交易活动或者欺诈造假的行为时,可以非常容易地通过区块链中的数据快速识别这些非法行为。

## 1.6.3 区块链在公证领域的应用

公证是公证机构根据自然人、法人或者其他组织的申请,依照法定程序对民事法律行为,有法律意义的事实和文书的真实性、合法性予以证明的活动。公证最核心的一点是存证信息的完整性和抗篡改性。由于区块链天然具备数据不可逆不可篡改的特性,因此区块链技术是非常适合公证系统的。

### 1. 公正领域的行业痛点

传统公证存在手续繁琐、处理低效等痛点。在当前中心化系统框架下的中心数据库承受着日益增长的数据存储和安全维护的双重压力。在公证行业内部、公证行业与其他部门之间的信息沟通、信息共享和信息协作中存在交流不够充分等问题。同时公证行业的业务领域由于面临国家政策调整等原因,导致一部分原有业务下滑和费用调整。还有电子存证业务开



展受到第三方存证公司业务挑战,互联网公证也存在难以实现等问题。

## 2. 区块链在公证领域中的应用

区块链具有信息不可篡改、数据加密保存、所有节点保存完整副本等特点。这些特点具有天然帮助解决传统公证行业的困难和业务新诉求的属性。首先,区块链的去中心化决定了它的安全性和可扩展性,公证数据存储的难题在区块链技术下迎刃而解。其次,区块链可以提高公证业务网络服务平台的信息流畅性,在对个人隐私进行保护的同时提高办证效率。同时,区块链能从技术上保障在线和远程监管办证的质量。区块链还能实现有效监管公证业务的质量,做到责任追溯时有据可依。最后,区块链能够推进信息互联共享,进一步加强公证处与外部机构的协调沟通,为办理公证业务提供有效的信息核实手段等。通过区块链这个“创造信任的机器”结合原有被国家法律赋予的国家公信力可以达到“双信合一”,实现“政策+科技”的双重增信。

## 3. 区块链在公证行业的应用场景

### 应用场景 1: 证书公证

区块链技术可以有效地解决证书公证中存在的问题。以学历证书为例,在应聘、考评等情况下,需对学历或所持毕业证书(学位证书)的真实性、合法性予以证明。尤其是在涉外学历证书时,对证书公证的需求更为频繁。但是目前中心化的证书验证平台给证书的伪造和篡改提供了可乘之机。如果将证书信息存放在区块链中,利用区块链的数据加密存储和数据不可逆的特性可以有效防止证书被篡改。同时由于区块链分布式总账的特性,相关合作机构可以通过部署节点的方式同步数据,这样即防止了相关机构篡改证书同时还可以提高证书的访问效率。

### 应用场景 2: 法律证据公证

对于经济体而言,每一份合同都可能成为日后的重要证据,对合同进行公证将极大有利于其法律权益的伸张。对于个人而言,取得关键法律证据的公证成为保护自身合法权益的关键,例如遗嘱公证以及语音、邮件、微信、微博等各种类型的法律证据,都是法律申诉的有力证据。律师作为专业法律咨询服务提供者,单位时间的效率十分重要,但往往由于“取证难”而耗费大量宝贵时间。如果有一种便利、简单的取证工具,将极大有利于个体合法权益的保护以及提升律师工作效率。区块链天生具有解决这些问题的属性。区块链中数据的防篡改和不可逆的特性可以有效避免伪造证据,同时由于区块链中的数据是所有参与方共同维护的,因此可以让所有参与方共同维护同一份证据链,这样能有效地避免单个证据的孤立性和证据链的断裂。

## 1.6.4 区块链在数字版权领域的应用

随着互联网特别是移动互联网的发展,数字出版已经形成较为完整的产业链,通过相



关产业链网络作家可以获取可观的收入。但是目前数字版权的保护并没有得到很好的发展,数字内容的盗版现象非常严重。区块链技术的出现为数字版权的发展带来契机。

### 1. 数字版权行业的痛点

随着知识经济的兴起,知识产权已成为市场竞争力的核心要素。互联网是知识产权保护的前沿阵地,但当下的互联网生态里知识产权侵权现象非常严重,网络著作权官司纠纷频发。这些问题严重侵蚀原创精神,同时针对盗版问题也存在举证困难、维权成本过高等客观因素。因此侵权和盗版问题成为内容产业的尖锐痛点。侵权盗版制约着相关行业的进一步发展,同时各参与方都深受其害,其中作者等内容生产方一直处于弱势地位,缺少相应的话语权和主导权,创作积极性倍受打击。面对这些问题国家非常重视,各种政策和扶持计划频出,重拳解决版权保护难题。但是限于技术手段很难从根本上解决。

### 2. 区块链在数字版权中的应用

区块链基于数学原理解决了交易过程中所有权确认的问题,在价值交换活动中产生的记录及其记录都是可信的。区块链记录的信息一旦生成将永久记录并且无法篡改,除非能拥有全网络总算力的51%以上,才有可能修改最新生成的一个区块记录。这些特性可以保证数字版权的拥有者能够支配自己的智力成果。同时还可以借助区块链的账本属性在区块链中直接将自己的数字资产进行交易。

### 3. 区块链在数字版权的应用场景

#### 应用场景1: 图书出版

区块链应用在图书出版行业中可以有效地保护图书的版权信息。图书在出版之前可以在区块链版权系统中进行相关版权信息的登记即确定了作品归属,相当于为原创内容登记了一张“数字身份证”。这样从源头保护原创版权,为后续的维权以及版权变现等需求打下良好的基础。

#### 应用场景2: 音乐创作

音乐行业的市场规模巨大,但在传统模式下音乐人很难获得合理的收益。利用区块链技术使音乐整个生产和传播过程中的收费和用途都是透明并且真实的,这样能有效确保音乐人直接从其作品的销售中获益。另外,音乐人跨出出版商和发行商,通过区块链平台自行发布和推广作品,不但不需要担心侵权问题还能更好地管理自己的作品。

## 1.6.5 区块链在保险行业的应用

近年来借助互联网的东风,保险行业得到了快速的发展,但是在高速发展的同时由于互联网天生的缺陷也带来了诸如欺诈、骗保等负面影响。保险行业在呼唤新技术能够解决这些问题。区块链的出现在某种程度上让保险行业有了二次腾飞的翅膀。

### 1. 保险行业的痛点

保险行业中保险公司要接触大量的 C 端用户, 会花费大量的时间和精力收集和甄别客户信息。这些步骤直接导致了用户身份认证非常困难。目前的保险数据都是采用中心化数据库的存储方式, 因此存在单一节点易被控制等隐患。保险公司在承保和理赔的过程中掌握了客户大量诸如身份、医疗健康等敏感信息, 这些信息一旦泄露会给保险公司和投保人带来非常严重的后果。

### 2. 区块链在保险行业中的应用

区块链技术的安全、信任、自动化、可追溯性等特点可以应用于保险行业的承保管理、运营风险管控、客户服务、信息安全、保险反欺诈等领域。同时区块链技术也给保险行业在商业模式创新等方面提供了一个不同的视角和全新的实现路径。区块链将成为保险创新的新动力。首先, 区块链技术可以数字化管理个人数据, 精简的数字认证, 通过区块链技术, 保险公司与个人之间可以建立更直接、更有效的关系。其次, 区块链能够进一步打破不同地域的地理隔阂, 让保险的覆盖率可以从空间上进行调整, 积极推动了金融的包容性。最后, 区块链技术的出现可以促进合约自动化的进程, 通过使用智能合约来实现效率的提升并使某些保险产品随着时间的推移实现自我管理。

### 3. 区块链在保险行业的应用场景

区块链应用在保险行业中可以优化保险业务流程, 助力保险服务体验升级。保险公司可提供用户信息, 这些信息经过审查验证后写入区块链, 购买不同保险时无须重复输入个人信息, 在区块链上查询即可, 这能大大缩短投保时间。区块链技术还可以实现自动理赔。利用区块链的智能合约技术可以将理赔条件编写在智能合约中, 一旦达到特定出险条件, 即可快速理赔。

利用区块链的共享账本特性可以加强行业信息共享, 降低保险机构运营成本。利用区块链开源、透明的特点, 可构建各保险机构为节点的联盟区块链, 实现保险业信息的有效共享。例如在共保或再保情形下, 保险事件发生后合同相关的所有保险人、再保险人、承保代理人均希望跟进理赔流程并开展谈判。若通过搭建区块链, 将理赔文件编写入区块链, 所有成员机构均能监测到理赔进展并参与更新, 这样不仅能保证文件准确度更能极大缩短理赔时间, 降低运营成本。

通过区块链还可以构建信用机制和安全体系, 服务反保险欺诈和反洗钱等工作。区块链可追溯且不可篡改的特性, 在反保险欺诈、反洗钱等领域将具有广泛应用。比如, 可构建被保险人医疗信息区块链, 经过授权的医院或医生把病人医疗信息写入区块链, 保险公司在核保时通过查询被保险人的相应医疗信息, 可避免带病投保、虚假赔案等欺诈行为。



### 1.6.6 区块链在公益慈善领域的应用

随着互联网技术的发展,社会公益的规模、场景、辐射范围及影响力得到空前扩大。“互联网+公益”、普众慈善、指尖公益等概念逐步进入公益主流。这些模式不仅丰富了传统慈善的捐赠方式,同时推动了公众的公益行为向碎片化、小额化、常态化方向发展。同时,各式各样的公益项目借助互联网实现丰富多彩的传播,使公益的社会影响力被成百倍地放大。

#### 1. 公益慈善领域的行业痛点

慈善机构要获得持续支持,就必须具有公信力,而信息透明是获得公信力的前提。公众关心捐助的钱款、物资发挥了怎样的作用。既要知道公益机构做了什么,也要知道花了多少,成本有多高。这种公信度的高低和公益的成效决定了公益机构能否获得公众的认同和持久支持。然而,在过去几年里公益慈善行业爆出的一些“黑天鹅”事件,极大地打击了民众对公益行业的信任度。公益信息不透明不公开,是社会舆论对公益机构、公益行业的最大质疑。公益透明度影响了公信力,公信力决定了社会公益的发展速度。信息披露所需的人工成本,又是掣肘公益机构提升透明度的重要因素。

#### 2. 区块链在公益慈善领域中的应用

区块链从本质上来说是利用分布式技术和共识算法重新构造的一种信任机制,是用公信力助力公信力。区块链上存储的数据高可靠且不可篡改,天然适合用在社会公益场景。公益流程中的相关信息,如捐赠项目、募集明细、资金流向、受助人反馈等,均可以存放于区块链上,在满足项目参与者隐私保护及其他相关法律法规要求的前提下,有条件地进行公开公示。

为了进一步提升公益透明度,公益组织、支付机构、审计机构等均可加入进来作为区块链系统中的节点,以联盟的形式运转,方便公众和社会监督,让区块链真正成为“信任的机器”,助力社会公益的快速健康发展。

区块链中智能合约技术在社会公益场景也可以发挥作用。对于一些更加复杂的公益场景,比如定向捐赠、分批捐赠、有条件捐赠等,就非常适用智能合约来进行管理,使得公益行为完全遵从预先设定的条件,更加客观、透明、可信,杜绝过程中的猫腻行为。

#### 3. 区块链在公益慈善领域的应用场景

区块链与公益的结合,有丰富的应用场景和想象空间,目前已经有真实的应用案例投产上线。2016年7月,支付宝与公益基金会合作,在其爱心捐赠平台上设立了第一个基于区块链的公益项目,为听障儿童募集资金,帮助他们“重获新声”。在这次的项目中,捐赠人可以看到一项“爱心传递记录”的反馈信息,在进行了必要的隐私保护基础上,展示了自



己的捐款从支付平台划拨到基金会账号,以及最终进入受助人指定账号的整个过程。通过区块链技术既保障了公益数据的真实性,又能帮助公益项目节省信息披露成本,充分体现出了区块链公益的价值。

### 1.6.7 区块链与智能制造

智能制造(Intelligent Manufacturing, IM)是一种由智能机器和人类专家共同组成的人机一体化智能系统,它在制造过程中能进行智能活动,诸如分析、推理、判断、构思和决策等。通过人与智能机器的合作共事,去扩大、延伸和部分取代人类专家在制造过程中的脑力劳动。在智能制造的过程中不存在传统制造中的设计图纸,所有的制作标准高度数字化。但是,另一方面数字化也带来了其他结果。在缺乏强大的数据保护框架之下,数字化的参数在传递的过程中存在数据失窃和被篡改的可能性。区块链技术能有效地避免这些问题的发生。

#### 1. 智能制造行业的痛点

实施智能制造的重点任务就是要实现制造企业内部信息系统的纵向集成,以及不同制造企业间基于价值链和信息流的横向集成,从而实现制造的数字化和网络化。在现实中由于制造设备和信息系统涉及多个厂家,原本中心化的系统主要采用人工或中央电脑控制的方式,实时获得制造环节中所有信息的难度很大。同时所有的订单需求、产能情况、库存水平变化以及突发故障等信息都存储在各自独立的系统中,而这些系统的技术架构、通信协议、数据存储格式等各不相同。这些因素都严重影响了互联互通的效率,也制约了智能制造在实际生产制造过程中的应用。

#### 2. 区块链在智能制造领域中的应用

利用区块链技术,可有效采集和分析在原本孤立的系统中存在的所有传感器和其他部件所产生的信息,并借助大数据分析,评估其实际价值。通过这些数据可以对后期制造过程进行预期分析,从而帮助企业快速有效地建立更为安全的运营机制。数据透明化使研发审计、生产制造和流通更为有效,同时也为制造企业降低了运营成本和制造成本,提升了良品率,使企业具有更高的竞争优势。智能制造的价值之一就是重塑价值链,而区块链有助于提高价值链的透明度、灵活性,并能够更敏捷地应对生产、物流、仓储、营销、销售、售后等环节存在的相关问题。

#### 3. 区块链在智能制造的应用场景

##### 应用场景 1: 组建和管理工业物联网

组建高效、低成本的工业物联网,是构建智能制造网络基础设施的关键环节。在传统的组网模式下,所有设备之间的通信必须通过中心化的代理通信模式实现,设备之间的连接必须通过网络。这种模式极大提高了组网成本,同时系统的可扩展性、可维护性和稳定性都

不是很好。区块链技术利用 P2P 组网技术和混合通信协议有效处理异构设备间的通信问题,将显著降低中心化的数据中心建设和维护的成本。同时区块链技术可以将计算和存储需求分散到组成网络的各个设备中,有效阻止了网络中任何单一节点的失败而导致整个网络崩溃的情况发生。另外,区块链中分布式账本的防篡改特性能有效防止工业物联网中任何单节点设备被恶意攻击和控制后带来的信息泄露和恶意操控风险。

#### 应用场景 2: 生产制造过程的智能化管理

在传统的生产模式下,设备的操作、生产和维护记录是存储在单一、孤立的系统中,一旦出现安全和生产事故,企业、设备厂商和安全生产监管部门难以确保记录的真实性和一致性,也不利于后续事故的防范及设备的改进。利用区块链技术能够将制造企业中的控制模块、系统、通信网络、ERP 等相关系统连接起来并通过统一的账本,让企业、设备厂商和安全生产监管部门能够长期、持续地监督生产制造的各个环节,从而提高生产制造的安全性和可靠性。同时,区块链账本记录的可追溯性和不可篡改性也有利于企业审计工作的开展,便于发现问题、追踪问题、解决问题以达到优化系统的目的。

### 1.6.8 区块链在教育就业中的应用

教育就业作为社会文化传授、传播的窗口,需要实现学生、教育机构以及用人单位之间的无缝衔接,以提高教育就业机构的运行效率和透明度。区块链系统的透明化、数据不可篡改等特征,完全适用于学生征信管理、升学就业、学术、资质证明、产学合作等方面,对教育就业的健康发展具有重要的价值。

#### 1. 教育行业的痛点

在教育就业行业中由于学生信用体系不完整导致未能建立学生相关信息的历史数据信息链和相关的维度。这些问题导致政府、企业无法获得完整有效信息,从而直接导致学生无法便捷、公平地享受应有的服务。在就业过程学历造假、论文造假、求职简历造假等现象的存在直接使得用人单位、院校缺乏有效的验证手段,进而降低了学校与企业间、院校与院校间的信任度。除此之外,一些学术性实验、跨校组织的公开课以及多媒体教学资源等在网络上产生版权和学术纠纷,这些纠纷对学者以及研究人员产生了消极影响,进一步削弱了高等学府对学术研究的积极性。

#### 2. 区块链在教育就业中的应用

利用区块链技术对现存运行方案的不足之处进行优化,能有效简化流程和提高运营效率,并且能及时规避信息不透明和容易被篡改的问题。利用分布式账本记录跨地域、跨院校的学生信息可以方便追踪学生在校时期所有正面以及负面的行为记录,能帮助有良好记录的学生获得更多的激励措施,并构建起一个良性的信用生态。利用区块链技术可为学术成果



提供不可篡改的数字化证明,为学术纠纷提供权威的举证凭据,降低纠纷事件消耗的人力与时间。同时,这种数字化证明可以与已有的应用无缝整合,为每一个文字、图片、音频、视频加盖唯一的时间戳身份证明,交叉配合生物识别技术,从根本上保障了数据的完整性、一致性以达到保护知识产权的目的。

### 3. 区块链在教育就业的应用场景

#### 应用场景 1: 教育存证

在教育存证领域,基于区块链的学生信用平台可创建含有基本信息的数字文件,然后使用用户的私钥对证书的内容进行签名,再对证书本身附加签名。依赖于生成的哈希值,可以验证证书内容是否被篡改。最后,再用私钥在区块链上创建一条数字记录,保证用户信息和证书内容的一致性。教育机构利用自己的私钥签署一份具有完整信息记录的数字证书,将其哈希值存储在区块链中,在每一次发放和查询时都会由智能合约触发相应的多重签名校验以确保记录不会被恶意查询,交易输出时将数字证书分配给需求方,如学生或者用人单位。

#### 应用场景 2: 产学合作

产学合作是教育机构与用人单位之间多赢的机制,现在教育存在的问题之一就是封闭办学,即学生的技能信息、知识体系未与用人单位的技能需求、市场趋势保持信息对称。通过引入区块链技术实现学生技能与社会用人需求无缝衔接,可精确评估人才录用、岗位安排的科学性和合理性,有效促进学校和企业之间的合作。

## 1.7 区块链的其他常见技术框架

区块链技术从诞生至今,除了比特币之外,相关技术社区根据区块链技术原理已经实现了很多技术框架。在上文提到了区块链技术框架中极具代表性的三个技术框架,比特币、以太坊、超级账本。这三个技术框架会在后面章节中重点介绍,这里不再复述。除了这三个典型的技术框架之外,还有许多基于区块链基本原理实现的技术框架。这些框架都有自身的特点,每个框架解决问题的侧重点也是不一样的,我们选择其中比较有代表性的介绍给读者以供参考。

### 1. Corda

Corda 是由 R3CEV 推出的一款分布式账本平台,其借鉴了区块链的部分特性,例如 UTXO 模型以及智能合约,但它在本质上又不同于区块链,并非所有业务场景都可以使用这种平台,Corda 面向的是银行间或银行与其商业用户之间的互操作场景。

项目地址如下所示:

<https://github.com/corda/corda>



## 2. BigChainDB

BigChainDB 填补了去中心生态系统中的一个空白，是一个可用的去中心数据库。它具有每秒百万次写操作、存储 PB 级别的数据和亚秒级响应时间的性能。BigChainDB 的设计起始于分布式数据库，通过创新加入了很多区块链的特性，如去中心控制、不可改变性、数字资产的创建和移动。

BigChainDB 继承了现代分布式数据库的特性：吞吐量和容量都是与节点数量线性相关，功能齐全的 Nosql 查询语言，高效的查询和权限管理。因为构建在已有的分布式数据库上，它在代码层面也继承了企业级的健壮性。可扩展的容量意味着具有法律效力的合同和认证可以直接存储在区块链数据库里。权限管理系统支持从私有企业级区块链数据库到开放公有的区块链数据库配置。BigChainDB 是对于像以太坊这样的去中心处理平台和 Inter Planetary File System (IPFS) 这样的去中心文件系统的补充。

项目地址如下所示：

<https://github.com/bigchaindb/bigchaindb>

## 1.8 本章小结

本章的目的是让不了解区块链的读者能对区块链有一个宏观的认识，主要介绍了区块链的起源和演进路线、区块链的核心技术及其特性、区块链技术的缺点和人们对它的错误认识，以及区块链在不同行业的应用。

## 实战准备

区块链技术是由多种技术组合而成的一个技术栈，要正确了解和掌握区块链技术需要对整个区块链技术栈有一个初步的了解。本章会介绍区块链系统中经常用到的一些技术和工具，这些工具在使用区块链技术的时候经常被用到。在作者运营区块链技术论坛的过程中，通过对数百个问题进行分析之后我们发现，很多区块链技术问题都不在区块链本身，而是由于对区块链周边一些技术的不熟悉而引起的。我们建议读者在阅读本书后续内容之前，首先仔细阅读本章内容。

### 2.1 开发环境准备

学习和使用区块链技术，除了要对区块链平台的开发语言有所了解之外，还需要了解操作系统、虚拟机软件等相关技术，这些技术在区块链的使用过程中占有比较重要的比重。我们这套系列书的实战项目主要是基于 Fabric、以太坊、比特币这三个平台。本节内容将重点介绍这三个区块链主流技术平台对操作系统的要求。

#### 2.1.1 操作系统的配置

目前主流的区块链平台基本上都支持 Linux、MacOS、Windows 这三个常用的操作系统。但是由于各个平台本身的特性，我们建议在 Linux 或者 MacOS 系统中部署和测试区块链平台。Linux 可以选择 CentOS 或者 Ubuntu 这两个平台。如果平时比较习惯 Windows 系统，那么我们建议另外准备一台安装 CentOS 或者 Ubuntu 的机器。如果没有条件，可以选择虚

拟机器软件 xbox 或者 vmware 来安装一台虚拟的操作系统。

本书是一本实战性很强的书，书中涉及大量的实际操作过程。因此在开始阅读本书后续章节之前，我们建议读者先准备相关的硬件，建议配置如下：

- 一台 MacOS 系统的电脑
- 一台 Ubuntu 系统的电脑
- 一个 Windows 系统的机器 + 一台安装 CentOS 或者 Ubuntu 的实体机（如果条件不具备，通过虚拟机软件安装 CentOS 或者 Ubuntu 也是可以的）



**注意** 在一台安装有 Windows 系统的电脑上面运行本书后面的例子时可能会遇到问题。

在学习环境中可以按照上面的配置，但是在生产环境中一般都将区块链系统部署在 Linux 平台之上，可以是 CentOS 或者 Ubuntu。各个区块链平台对操作系统的版本要求是不一致的，在后续章节中我们将详细介绍各个区块链平台对操作系统的要求。

## 2.1.2 Docker 的使用

Docker 是一个开源的应用容器引擎，它让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化，容器完全使用沙箱机制，相互之间不会有任何接口。Docker 在区块链相关的技术平台中应用非常广泛，特别在 Fabric 中几乎无法离开 Docker。本节中我们将给大家介绍 Docker 的安装方式以及几个常用的 Docker 命令。

### 1. Docker 的安装和配置

#### (1) Ubuntu 上面安装 Docker

执行下面的命令安装 Docker 和 Docker Compose：

```
apt install docker.io
apt install docker-compose
```

安装完成之后系统会自动启动 Docker。

#### (2) CentOS 上面安装 Docker

由于 CentOS 在启动程序的时候有所不同，有的版本采用 service 命令启动，有的版本采用 systemctl 命令，为此我们在可能出现重复的地方同时标注两个命令，各位读者在进行实践操作的时候要注意。

执行下面的命令：

```
yum install docker python-pip
pip install --upgrade pip
```



```
pip install docker-compose
```

安装完成之后需要执行下面的命令启动 docker:

```
service docker start
```

docker 安装完成之后执行以下命令检查是否打开 http 通道, 否则后面会出现无法识别的错误。

```
curl -XPOST --unix-socket /var/run/docker.sock -d '{ "Image": "nginx" }' -H
'Content-Type: application/json' http://localhost/containers/create
```

## 2. Docker 的常用命令

### (1) 查看 Docker 版本

```
docker version**
```

### (2) 查看镜像的列表

```
docker images
```

### (3) 查看正在运行的镜像

```
docker ps
```

### (4) 查看容器的详细信息

```
docker inspect 142aef786231
```

docker inspect 后面的参数是 docker ps 命令返回结果中的 CONTAINER ID 字段的值, 这是一个非常重要的命令。

### (5) 运行镜像

```
docker start -it -p 8880:80 nginx
```

docker start (start 后面的参数为容器编号或者容器名称, 在执行完 docker create 命令之后, 可以通过 docker ps -a 命令获取)。

### (6) 创建容器并且运行容器

```
docker run -d -p 8880:80 nginx
```

- -d: 守护进程运行。
- -p: 端口映射, 后面的是 Docker 容器内的端口, 前面的是宿主服务器的端口。
- nginx: 是镜像的名字。

### (7) 查询本镜像

```
docker images
```

### (8) 停止正在运行的镜像实例(容器)

```
docker stop 0067a3c9ef6c
```



**注意** stop 后面的参数是镜像运行实例的编号, 可以支持运行多个镜像的实例(多个容器), 镜像实例的编号可以通过命令 `docker ps` 获取, CONTAINER ID 就是实例的编号。

### (9) 进入容器

```
docker exec -it c4dbfde3b039 /bin/bash
```

其中 -it 后面的就是容器实例的编号, 获取实例编号的方法上面已经说过了。

### (10) 导出 Docker 镜像

```
sudo docker save -o 导出的文件路径 镜像名称:版本号
```

### (11) 导入 Docker 镜像

```
docker load --input 镜像文件名
```

### (12) 查询 hub.docker.com 中的镜像

```
docker search
```

### (13) 下载 hub.docker.com 中的镜像

```
docker pull nginx
```

### (14) 删除本地已经存在的镜像文件

```
docker rmi 镜像名称或者镜像编号
```



**注意** 如果直接用, 可以在镜像名字的后面加上 :+ 标签号。

### (15) 停止容器并删除

```
docker kill $(docker ps -q)&&docker rm $(docker ps -qa\)
```

停止所有正在运行的 Docker 容器, 然后删除所有 Docker 容器文件。这两个命令可以拆开运行。

### (16) 根据关键字删除 Docker 镜像

```
docker rmi -f $(docker images\|grep $keyword\|awk '{print $3}')
```

删除已经下载的 Docker 镜像, 上述命令中的参数 \$keyword 为所删除文件中包含的关

键字。比如 \$keyword 的值为 "fabric", 那么上述命令将会删除名称中包含 "fabric" 字符串的镜像文件。

### 2.1.3 Git 的使用

Git 是一个开源的分布式版本控制系统, 用于敏捷高效地处理任何或小或大的项目。Git 与常用的版本控制工具 CVS、Subversion 等不同, 它采用了分布式版本库的方式, 不需要服务器端软件支持。Git 在没网的时候仍然可以使用大部分命令信息操作, 在网络恢复的时候再跟服务器进行同步, 这样可以更好地实现多人联合编程。

目前几乎所有的区块链项目都是开源项目, 这些开源项目的源代码都是托管在 github 上面的, 了解常用的 Git 命令对阅读本书特别是操作本书提供的示例还是有帮助的。下面我们将介绍常用的几个 Git 命令。

#### 1. 将远程的版本库克隆到本地

```
git clone
```

#### 2. 获取当前版本库的状态

```
git status
```

#### 3. 提交代码

```
git add . 或者 git add 文件名  
git commit -m 注释  
git push
```

#### 4. 从远程版本库获取最新的代码更新

```
git pull
```

## 2.2 开发语言

本书介绍的区块链技术中主要涉及 Golang、Node.js 等编程语言。了解这些编程语言的特性对学习和使用相关的区块链技术平台是有帮助的。

### 2.2.1 GO 语言

Go 语言又称为 Golang (本书中一律称为 Golang), 是谷歌 2009 年发布的第二款开源编程语言。Go 语言专门针对多处理器系统应用程序的编程进行了优化, 使用 Go 编译的程序可以媲美 C 或 C++ 代码的速度, 而且更加安全, 支持并行进程。目前大多数区块链项目都是用 Go 语言开发的 (如以太坊、Fabric)。但比特币由于出现年份稍早, 因此采用 C++ 语言



开发，所以针对比特币的项目需要了解以下 C++ 语言。

Go 语言是区块链平台的开发的重要语言。在学习区块链平台使用的过程中，为了更好地理解这些平台的技术特性有时候需要阅读相关项目的源代码，熟练掌握 Go 语言的基本语法特性是必要的。由于本书篇幅的限制，我们只列出对本书项目有用的一些特性供大家参考。

### 1. Go 语言的路径

Go 语言的包路径是 Go 语言中比较容易出错的部分，很多初学者在这个步骤容易出错。要彻底搞清楚 Go 语言的包路径问题可能需要参考专门的资料，这里不再详细说明。我们建议读者把所有的 Go 项目包括 Fabric 源代码和所有 Chaincode 代码都存放在 GOPATH 设定的路径下面。

下面是我编写的 GOPATH 路径和项目结构的示例。如果你对 Go 语言比较熟悉可以跳过这部分内容直接阅读后面的内容。

在命令行中执行命令 `go env`，结果如下所示：

```
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/project/goworkspace"
```

从上面的信息中可以看出作者的 GOPATH 为文件夹 `/project/goworkspace`。通过 `tree` 命令可以获取该文件夹中项目的结构如下：

```
├── github.com
│   ├── 17golang
│   ├── golang
│   ├── hyperledger
│   ├── satori
│   └── xuehuiit
├── golangstudy
│   ├── main
│   ├── study
│   └── utils
├── qklszzl
└── chaincodestudy
```

这里将所有项目都存放在 GOPATH 所设定的目录中的 `src` 文件夹中，也许不是最好的项目结构方式，但是考虑到读者有很多人是初次接触 Golang，这种方法能减少出错的几率。如果读者对 Golang 比较熟悉，可以选择自己熟悉的项目结构方式。

## 2. Golang 的 IDE 工具

Goland 是 Golang 集成开发环境, 提供了代码提示、语法错误提示、程序调试等功能。可以通过下面的网址下载:

<https://www.jetbrains.com/go/>

### 2.2.2 Node.js

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型, 使其轻量又高效。Node.js 的包管理器 npm, 是全球最大的开源库生态系统。Node.js 这门语言本身和区块链的关系不是很大, 但是很多优秀的基于区块链的开源项目都是基于 Node.js 开发的, 这些应用对我们学习和使用区块链的技术提供了很好的借鉴作用, 有时候我们需要阅读这些项目的源码来验证我们的想法。

本书所有的实战项目也都是用 Node.js 开发的, 因此了解 Node.js 相关的特点有助于更好地阅读与理解本书的内容。Node.js 使用的是 Javascript 的语法特性, 如果你对 Javascript 不是非常熟悉可以参考相关的资料。

Node.js 的 IDE 我们推荐使用 webstorm。webstorm 是功能丰富的 Node.js 集成开发环境, 提供了代码提示、调试器等常用功能。可以通过以下路径下载:

<https://www.jetbrains.com/webstorm/>

## 2.3 常用工具

### 2.3.1 Curl

Curl (Command Line URL Viewer) 是一个 Linux/Windows 命令行工具。Curl 能从服务器下载数据, 也能往服务器上发送数据。Curl 支持多种协议, 包括: DICT、FILE、FTP、FTPS、GOPHER、HTTP、HTTPS、IMAP、IMAPS、LDAP、LDAPS、POP3、POP3S、RTMP、RTSP、SCP、SFTP、SMB、SMBS、SMTP、SMTPS、TELNET 和 TFTP。从 Curl 支持的协议就可以看出, CURL 命令非常强大。在区块链系统中, 有些时候需要通过 Curl 来快速调用相关的 JSON-RPC 接口, 以便快速完成相关的操作。

由于本书篇幅有限, 无法详细介绍 Curl 所有的命令, 并且在区块链系统中只需用到 Curl 中的部分命令, 因此本节中只简单介绍 Curl 在各个平台上的安装, 在后续章节中我们会在需要的时候详细地介绍相关的命令。

Curl 工具 CentOS 中的安装:

```
yum install curlapt install curl
```

Curl 工具 Ubuntu 中的安装:

```
apt install curl
```

### 2.3.2 tree

tree 命令是 Linux/UNIX 系统中常用的命令，可以非常方便地查看文件夹的结构，并且以树形目录的形式展现。命令执行结果示例如下所示：

```
.
├── ordererOrganizations
│   └── robertfabrictest.com
│       ├── ca
│       ├── msp
│       ├── orderers
│       ├── tlsca
│       └── users
└── peerOrganizations
    ├── org1.robertfabrictest.com
    │   ├── ca
    │   ├── msp
    │   ├── peers
    │   ├── tlsca
    │   └── users
    ├── org2.robertfabrictest.com
    │   ├── ca
    │   ├── msp
    │   ├── peers
    │   ├── tlsca
    │   └── users
    └── org3.robertfabrictest.com
        ├── ca
        ├── msp
        ├── peers
        ├── tlsca
        └── users
```

tree 命令的安装非常简单，tree 命令 Ubuntu 和 CentOS 的安装方式分别如下所示。

tree 命令 Ubuntu 中的安装:

```
sudo apt-get install tree
```

tree 命令 CentOS 中的安装:

```
yum install tree
```

### 2.3.3 Jq

Jq 是 Linux 处理 JSON 文件的工具，Jq 是一个基于命令行的工具，通过简单的命令可



以完成对 JSON 格式文件的操作, 比如取值、设值等操作。Jq 工具在 Ubuntu 和 CentOS 中的安装方式分别如下所示。

Jq 工具在 Ubuntu 中的安装:

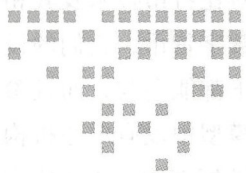
```
apt-get install jq
```

Jq 工具在 CentOS 中的安装:

```
yum install jq
```

## 2.4 本章小结

本章的内容看起来和区块链及技术关系不大, 但是在区块链技术的实际使用中还是比较重要的作用。虽然这些小技术不是什么复杂而宏大的技术, 但是对它们任何一个忽略都会会在后续的实际操作中带来障碍。我们可以躲避大树, 但是尝尝被小石头绊倒。通过阅读本章的内容可以有效地避免这些小障碍。



## Hyperledger 简介

本章主要介绍 Hyperledger 项目的背景、初衷、应用范围、项目结构等基本信息。通过阅读本章内容读者能够清晰地了解 Hyperledger 项目的来龙去脉和 Hyperledger 项目的结构体系。

### 3.1 Hyperledger 综述

Hyperledger（中文名为超级账本，在本书后续章节中统称为 Hyperledger）是 Linux 基金会于 2015 年发起的推进区块链数字技术和交易验证的开源项目。Hyperledger 的目标是让成员共同合作、共建开放平台以满足来自多个不同行业各种用户的需求，同时能大大简化业务流程。Hyperledger 的创始成员有 IBM、Intel、思科等大公司。截至本书完稿时已经加入 Hyperledger 的机构和公司已经超过 183 个，并且还在高速增长中。

Hyperledger 项目成立之初 Linux 基金会已经收到了多个不同的代码库，包括 IBM 代码库（一定程度上受以太坊启发），还有 DAH（Bits of Proof 比特币代码库）和 Blockstream 代码库（是比特币代码库的扩展）。除此之外还有 Digital Asset 和 Ripple 等贡献的代码。随着行业的发展，单一的项目已经无法满足业务的需求，因此 Hyperledger 逐步由一个单一的项目发展成了一个项目组。目前 Hyperledger 已经不是某个具体的技术，而是代表一组区块链技术框架的集合。截至目前，Hyperledger 项目组中一共包含 9 个正式项目和 50 多个这些正式项目的相关模块。

#### 3.1.1 Hyperledger 的项目背景

自从 2009 年比特币诞生以来，许多企业和技术社区花费大量的人力和物力研究其底

层技术。随着对比特币技术及其相关技术的深入研究,人们发现不管是比特币还是后来的以太坊及其他数字货币,它们的设计是完全开放、去中心化和非授权的。任何人在没有确定身份的情况下都能参与,而且参与的代价只需要贡献一点时间完成计算周期就行。在区块链的比特币模型中没有中心机构来发放许可,因为网络是非授权的。在像比特币和以太坊这样的区块链框架中,由于需要无数的算力来完成 POW 算法,以达到整个网络的完全和稳定,因此基于这些技术架构的系统的运行是昂贵的。这些特性和现有的绝大多数体系都是冲突的,如果直接将这些特性移植到现有系统中会存在很大的问题。但是区块链具备的数据不可逆、不可篡改以及分布式对等网络、数据集体维护等特性在非数字货币领域存在大量的应用场景。这些矛盾的存在是对新技术出现的呼唤,在这种情况下 Hyperledger 应运而生。

Hyperledger 是对传统区块链模型的革新。Hyperledger 通过提供一个模型,这种模型在某种程度上允许创建授权的和非授权的区块链。除此之外,Hyperledger 通过一个提供针对身份识别、可审计及隐私的安全和健壮模型,使得缩短计算周期、提高规模效率和响应各个行业的应用需求成为可能。

Hyperledger 的愿景是借助项目成员和开源社区的通力协作,共同制定并建立一个开放、跨产业、跨国界的区块链技术开源标准。它通过创建通用的分布式账本技术,协助组织扩展,建立行业专属应用程序、平台和硬件系统来支持成员各自的交易业务。

Hyperledger 由于点对点网络的特性,分布式账本技术是完全共享、透明和去中心化的,故非常适合应用于金融行业。同时在诸如制造、银行、保险、物联网等无数个其他行业都有非常大的应用前景。Hyperledger 通过创建分布式账本的公开标准来实现虚拟和数字形式的价值交换,例如资产合约、能源交易、结婚证书等通过 Hyperledger 能够安全、高效和低成本地进行追踪和交易。

### 3.1.2 Hyperledger 的项目成员

Hyperledger 是 Linux 基金会为推动区块链科技而组成的联盟。早期加入的成员包括荷兰银行、埃森哲等不同利益机构。由于分布式账本能够高效并以低成本进行业务流程,因此许多企业都开始投入区块链研究并最终选择加入 Hyperledger。

Hyperledger 是一个全球合作的项目,截至本书完稿时一共有 183 个成员,其中核心董事会成员包括埃森哲、空中客车、美国运通、芝加哥商品交易所集团、戴姆勒、美国证券托管结算公司、富士通、日立、IBM、英特尔、百度金融、摩根大通等金融、医疗、物联网、航空及物流等领域巨头。可以说 Hyperledger 项目是区块链领域的超新星,用途非常广阔。

Hyperledger 在中国的发展势头非常值得期待。Hyperledger 的执行董事 Brian Behlendorf 说道,“目前为止,Hyperledger 中有四分之一的成员都是来自中国,可见中国对新科技是持



有正面的态度。但同时我们也鼓励来自世界各地更多的成员能够加入我们，不断壮大目前的区块链开源技术。开源计划能够加速区块链的发展，而 Hyperledger 正是一个代表。”

## 3.2 Hyperledger 的体系结构

前文提到，Hyperledger 不是一个单独的项目而是包含了多个子项目的项目组，Hyperledger 项目组目前一共包含了九个正式项目，每个正式项目都包含若干个模块。这些正式项目和模块都是开源项目，任何人都可以下载并进行学习和研究。如果是技术人员还可以申请成为 Hyperledger 相关项目或者模块的开发者，为 Hyperledger 的发展贡献自己的力量。下面我们将分别介绍如何成为 Hyperledger 项目的开发者和 Hyperledger 中项目及模块的构成情况。

### 3.2.1 获取 Hyperledger 源代码并成为开发者

#### 1. 下载 Hyperledger 相关项目的代码

Hyperledger 所有项目代码统一由 Linux 基金会负责维护。Linux 基金会提供了两种下载 Hyperledger 项目代码的方式：Linux 基金会官方网站和 Github。

如果需要从 Linux 基金会的网站下载 Hyperledger 项目的源代码，可以通过下面的域名：

```
https://gerrit.hyperledger.org/r/#/admin/projects/
```

如果需要从 Github 下载 Hyperledger 项目的源代码，可以通过下面的域名：

```
https://Github.com/hyperledger
```

通过上述方法可以获取 Hyperledger 中相关项目的源代码，这些源代码中都包含了相关的使用手册，这些手册中详细地说明了这些项目和模块的编译、安装、调用、维护的方法。

#### 2. 成为 Hyperledger 项目的开发者

如果有兴趣加入 Hyperledger 项目并且成为开发者，那么首先需要注册一个 Linux 基金会的账号，注册地址如下所示：

```
https://identity.linuxfoundation.org/
```

注册完成之后可以给 Hyperledger 相关项目的负责人发送邮件告诉他们你希望加入该项目或者模块，在获取管理员的许可之后，登录网站可以查看到相关的授权信息。查看授权信息的网址如下：

```
https://gerrit.hyperledger.org/r/#/settings/group-memberships
```

### 3.2.2 Hyperledger 的 9 个正式项目

#### 1. Hyperledger 正式项目和模块的关系

通过前面的内容我们知道 Hyperledger 中有 9 个正式项目，同时这些正式项目都包含若干个模块。这些正式项目和它们包含模块之间的关系如图 3-1 所示。

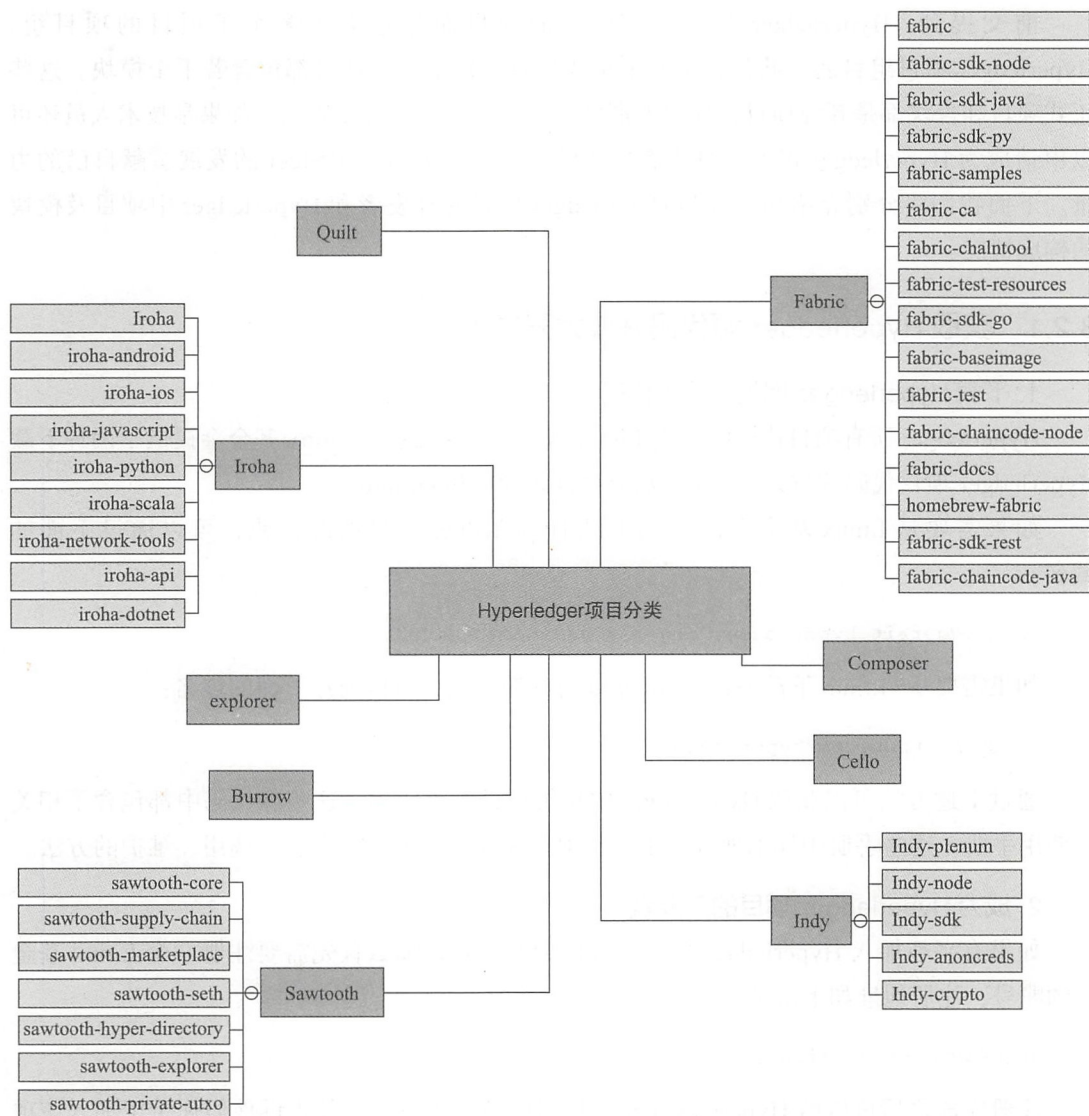


图 3-1 Hyperledger 项目模块关系图

图 3-1 中列举了 Hyperledger 项目组中的正式项目及其模块之间的关系。除了这些正式项目及其模块之外还有一些项目并没有出现在这张图中，因为那些项目是一些辅助性的项

目, 比如测试案例或者文档, 这里不再赘述。辅助项目的详细信息可以通过 Hyperledger 的 Github 网站获取。

## 2. Hyperledger 的正式项目

Hyperledger 的 9 个正式项目解决了区块链的核心基础问题, 比如分布式账本、区块链结构浏览器、不同区块链之间如何进行价值交换等。下面我们将重点介绍这 9 个正式项目的详细信息。

### (1) Hyperledger Fabric

Hyperledger Fabric 是 Hyperledger 的核心项目, 甚至在一些场合当提到 Hyperledger 的时候一般会认为就是指 Fabric, 这其实是误解, 但是也从侧面反映 Hyperledger Fabric 在 Hyperledger 中占据的地位。Hyperledger Fabric 本质上是一个分布式共享账本。Hyperledger Fabric 的目标是成为开发应用和解决方案的基础, 在设计上它采用模块化架构, 模块化架构的好处是组件可以根据需要灵活配置, 可以做到插入即用。Hyperledger Fabric 最初由 IBM 开发, 目前由整个 Linux 基金会共同维护。Hyperledger Fabric 是本书重点介绍的区块链技术框架之一, 关于 Hyperledger Fabric 的详细信息可以参考本书的第 4 ~ 13 章的内容。

### (2) Hyperledger Explorer

Hyperledger Explorer 是一个用来对区块链进行配置管理、区块和交易数据查询、节点管理的工具。通过 Hyperledger Explorer 可以查看区块链内部的信息, 比如: 账本数、区块数、交易数等数据。同时 Hyperledger Explorer 还可以对区块链进行管理, 比如执行部署智能合约、更新智能合约等常用操作。Hyperledger Explorer 的愿景是支持 Hyperledger 下面的所有区块链产品, 目前 Hyperledger Explorer 只支持 Fabric, 未来将逐步支持更多的区块链实现。Hyperledger Explorer 采用 Node.js 开发。

Hyperledger Explorer 由 IBM、Intel 和 DTCC 发起, 目前代码由 Linux 基金负责维护, 本书的两位作者目前是 Hyperledger Explorer 的主要代码维护者。

### (3) Hyperledger Iroha

Hyperledger Iroha 使开发者和 Hyperledger 之间的互动性更强, 当开发者需要使用分布式账本技术的时候 Iroha 会提供非常强大的帮助。Iroha 采用 C++ 开发, 基于领域驱动 C++ 设计, 在移动应用方面 Iroha 也提供了很好的支持。Iroha 目前采用新的 BFT 共识算法。

Hyperledger Iroha 项目由 Makoto Takemiya (Soramitsu)、Toshiya Cho (Hitachi)、Takahiro Inaba (NTT Data) 和 Mark Smargon (Colu) 几个人提出。目前项目已经提交给 Linux 基金会负责管理, 项目目前正处于孵化阶段。Iroha 项目的源代码被托管在 Github 中, 除了 Iroha 正式项目, 还有一些为 Iroha 提供服务的模块, 这些模块的源代码也被托管到 Github 中, 这些模块的基本信息如表 3-1 所示。



表 3-1 Hyperledger Iroha 项目相关模块源代码地址表

模块名称	模块项目地址	用 途
iroha	github.com/hyperledger/iroha-api	Iroha 项目核心代码
iroha-android	github.com/hyperledger/iroha-android	安卓版本的 Iroha
iroha-ios	github.com/hyperledger/iroha-ios	iOS 版本的 Iroha
iroha-javascript	github.com/hyperledger/iroha-javascript	Javascript 版本的 Iroha
iroha-python	github.com/hyperledger/iroha-python	Python 版本的 Iroha
iroha-scala	github.com/hyperledger/iroha-scala	Scala 版本的 Iroha
iroha-dotnet	github.com/hyperledger/iroha-dotnet	Dotnet 版本的 Iroha
iroha-network-tools	github.com/hyperledger/iroha-network-tools	网络工具
iroha-api	github.com/hyperledger/iroha-api	Iroha 的 API

(4) Hyperledger Burrow

Hyperledger Burrow 是 Hyperledger 中第一个源于以太坊框架的项目。Hyperledger Burrow 是一个经过许可的智能合同机。Hyperledger Burrow 发布于 2014 年 12 月，首次提供了一个模块化的、带有经过许可的智能合约解释器的区块链客户端。Hyperledger Burrow 采用了部分以太坊虚拟机（EVM）的技术规范。Hyperledger Burrow 项目包含以下模块：

- 共识引擎：负责维护节点之间应用程序引擎的事务排序以及网络堆栈。
- 应用程序区块链接口（“ ABCI”）：为共识引擎和智能合约引擎提供用于连接的接口规范。
- 智能合约引擎：为应用程序开发者提供了一个强大的智能合约执行引擎，可用于复杂的工业应用。
- 网关：为系统集成和用户界面提供编程接口。

Hyperledger Burrow 在设计上具有以下特点：

- 分布式账本技术采用模块化设计，可实现热插拔。
- 基于 EVM（以太坊虚拟机）规范，可以直接运行以太坊的智能合约，同时 EVM（以太坊虚拟机）被设计成无状态，这样设计的好处是很容易和其他区块链底层框架集成，同时可加载任意合约事务的应用状态。
- 智能合约引擎基于本地运行。同时智能合约执行环境需要授权，安全性比较好。在 Burrow 的路线图中将实现在 Hyperledger 项目中不同区块链产品之间的智能合约跨链调用。
- 提供了区块链应用接口（Application BlockChain Interface, ABCI），允许应用的拜占庭容错复制可以由任意一种编程语言编写。
- 共识引擎采用模块化设计，支持即插即用。共识方法的支持方面，目前除了支持 PBFT 还支持 SBFT 和 PoET，未来将支持更多的共识算法。

目前 Hyperledger Burrow 项目由 Monax 和 Intel 进行维护，由 Hyperledger 进行孵化，

在 Github 上托管代码，代码地址如下所示：

`https://github.com/hyperledger/burrow`

### (5) Hyperledger Indy

Hyperledger Indy 项目专注于区块链生态系统的数字身份工具，提供基于区块链或者其他分布式账本的数字身份，从而让它们跨管理域、跨应用与其他应用程序进行交互操作。

在传统的社会中每个人可能拥有多个身份，公司员工、车主、房屋所有人等。这些身份都存在于各自独立的系统中，互相之间没有联系。Hyperledger Indy 试图将这些身份信息统一处理，提供一套独立于任何系统的数字身份账本来解决身份统一管理的问题。

#### a) Hyperledger Indy 的节点类型

Hyperledger Indy 采用区块链技术作为其底层技术，在 Hyperledger Indy 中节点分为两种类型：验证节点和观察者节点。

- 验证节点：Hyperledger Indy 的验证节点主要用来处理客户端发起的写入请求，参与交易的共识。目前 Hyperledger Indy 计划在全球部署 120 个左右的节点。
- 观察者节点：Hyperledger Indy 的观察者节点主要负责处理客户端读取数据的请求，观察者节点需要从验证节点中同步状态数据。观察者节点可以通过提升自己的服务性能和增加公信力的方式升级为验证节点。目前 Hyperledger Indy 计划在全球部署数千个观察者节点。

#### b) Hyperledger Indy 的验证方法

Hyperledger Indy 通过其分布在全球的数千个节点来采集数据，根据内置的算法进行交叉验证，在涉及隐私保护的地方采用了零知识证明的加密技术。在请求速度上面，Hyperledger Indy 可以做到实时请求响应，无须直接联系身份持有者。Hyperledger Indy 采用多验证节点的方式从多个纬度获取验证信息，比如信用积分、负债、收入等。在隐私保护方面除了底层的零知识证明加密算法外，在应用层采用了匿名、有向图阻断、信息选择性披露等方法保护隐私。

#### c) Indy 的存储

Hyperledger Indy 与常见的区块链或分布式账本系统一样基于键值对（key-value）型数据库，其中 key 为 DID，value 存储各种与该 DID 关联的身份的描述信息。在实际应用中，value 存储的值可以包含验证请求的方法、相关信息的哈希值、DID 服务端描述（比如链接 Fabric 和以太坊，记录有关的 block 信息）等。

Hyperledger Indy 由 Sovrin 基金会牵头推进，Sovrin 基金会成立于 2016 年，致力于为去中心化的身份提供解决方案。目前 Hyperledger Indy 项目由 Linux 基金会管理，代码存放在 Hyperledger 的 Github 的项目组中。除了 Hyperledger Indy 核心项目，还有一些为 Hyperledger Indy 提供服务的模块，这些模块的源代码也被托管到 Github 中，这些模块的地址如表 3-2 所示。



表 3-2 Hyperledger Indy 项目相关模块源代码地址表

模块名称	模块项目地址	用 途
indy-crypto	github.com/hyperledger/indy-plenum	拜占庭容错协议实现
indy-node	github.com/hyperledger/indy-node	nodejs 版本的 Indy
indy-sdk	github.com/hyperledger/indy-sdk	Indy 的 sdk
indy-anoncreds	github.com/hyperledger/indy-anoncreds	匿名认证协议
indy-crypto	github.com/hyperledger/indy-crypto	Indy 的证书系统

### (6) Hyperledger Sawtooth

Hyperledger Sawtooth (中文名：锯齿湖) 是一个模块化平台，用以创建、部署和运行分布式账本。Hyperledger Sawtooth 包含诺韦尔共识算法，计时验证 (PoET)，它针对的是以最小的资源消耗处理大量的分布式验证器。Hyperledger Sawtooth 还有一个典型的特点就是可以运行以太坊的智能合约。Hyperledger Sawtooth 可以代替单个集中式数据库，分布式账本中的参与者为共享计算贡献资源，确保对分布式账本的状态的普遍一致。

Hyperledger Sawtooth 中包含了一种新的共识算法：计时验证 (POET)，又称为时间消失证明。POET 共识算法的大致思路是这样的，每个节点发布块之前都要从一个 enclave (在 Sawtooth 中它代表一个可信操作) 获取一个随机的等待时间，等待时间最短的率先发布块 (相当于被选为 leader)，其中的 enclave 是通过新型的安全 CPU 指令来实现的。enclave 支持两个函数 “CreateTimer” 和 “CheckTimer”，CreateTimer 用于从 enclave 中产生一个 timer，CheckTimer 会去校验这个 timer 是不是由 enclave 产生、是否已经过期。如果满足以上两个条件就会生成一个 attestation (凭证)。attestation 中包含的信息可以用来校验 certificate 是否是由该 enclave 产生并且已经等待了 timer 规定的时间。成为 leader 的概率与捐献的资源成比例的，因为是通用处理器而不需要定制矿机，所以参与的门槛就比较低，节点就会比较多，整个共识会更健壮。

Hyperledger Sawtooth 目前由 Hyperledger 项目组负技术指导，代码存放在 Hyperledger 在 Github 的项目组中。除了 Sawtooth 正式项目，还有一些为 Sawtooth 提供服务的模块，这些模块的源代码也被托管到 Github 中，这些模块的地址如表 3-3 所示。

表 3-3 Hyperledger Sawtooth 项目相关模块的源代码地址表

模块名称	模块项目地址	用 途
sawtooth-core	github.com/hyperledger/sawtooth-core	Sawtooth 核心代码
sawtooth-supply-chain	github.com/hyperledger/sawtooth-supply-chain	分布式组件
sawtooth-marketplace	github.com/hyperledger/sawtooth-marketplace	应用市场
sawtooth-seth	github.com/hyperledger/sawtooth-seth	和以太坊相关的子项目
sawtooth-hyper-directory	github.com/hyperledger/sawtooth-hyper-directory	Sawtooth 工具集合
sawtooth-sawtooth-explorer	github.com/hyperledger/sawtooth-explorer	Sawtooth 浏览器
sawtooth-private-utxo	github.com/hyperledger/sawtooth-private-utxo	Sawtooth 的 utxo 组件



### (7) Hyperledger Composer

Hyperledger Composer 是一个应用程序的框架，可以简化 Fabric 应用程序的创建、部署和使用。通过 Hyperledger Composer，开发者可以轻松地对业务资产、参与者以及事务建模，将这些模型变成可行的 Fabric 区块链应用。目前 Composer 项目的源代码被托管在 Github 中，除了 Hyperledger Composer 正式项目，还有一些为 Hyperledger Composer 提供服务的模块，这些模块的源代码也被托管到 Github 中，这些模块的地址如表 3-4 所示。

表 3-4 Hyperledger Composer 项目相关模块的源代码地址表

模块名称	模块项目地址	用 途
composer	github.com/hyperledger/composer	Composer 核心代码
composer-sample-networks	github.com/hyperledger/composer-sample-networks	网络示例
composer-sample-applications	github.com/hyperledger/composer-sample-applications	应用程序示例
composer-tools	github.com/hyperledger/composer-tools	Composer 工具
composer-sample-models	github.com/hyperledger/composer-sample-models	模型示例
composer-vscode-plugin	github.com/hyperledger/composer-vscode-plugin	vs 代码插件
composer-atom-plugin	github.com/hyperledger/composer-atom-plugin	atom 插件
composer-knowledge-wiki	github.com/hyperledger/composer-knowledge-wiki	知识库

### (8) Hyperledger Cello

Hyperledger Cello 是一个 Fabric 的集成管理工具。Hyperledger Cello 的目标是建立一种方式来创建、管理和终止区块链。Hyperledger Cello 项目的愿景是：能够兼容 HyperLedger 下的其他项目，包括 Fabric、Iroha 和 Sawtooth 等。

### (9) Hyperledger Quilt

Hyperledger Quilt 是一种支付协议，主要应用于 Hyperledger 下面的不同区块链产品之间进行价值的传递和转换。目前 Hyperledger Quilt 项目正处于发展中。

## 3.3 本章小结

本章主要介绍了 Hyperledger 的来龙去脉、应用范围和组成部分。通过本章内容读者可以了解到 Hyperledger 不是一个单独的项目，而是由多个项目组成的项目组。本章对 Hyperledger 的相关项目进行了简单的介绍，读者可以根据具体的应用场景选择适合的项目。

## Fabric 快速入门

Fabric 是 Hyperledger 项目组中的核心项目，本章将简单介绍 Fabric 的特性以及 Hyperledger 项目组中与 Fabric 相关的项目的基本属性，同时将重点介绍 Fabric 的编译、安装、使用等内容。通过阅读本章内容读者可以大概了解 Fabric 的组成部分和基本的使用方式，为后续内容的阅读打下基础。

### 4.1 Fabric 的技术特性

Fabric 是 Hyperledger（超级账本）项目组中的一个项目。从区块链技术的演进过程看，Fabric 属于区块链 3.0 的技术范畴之内。但是，Fabric 同时具有区块链 1.0 和 2.0 系统的特性，比如 Fabric 可以共享账本，具有智能合约，可以通过共识算法确保数据的安全。但是作为一个典型的区块链 3.0 技术平台，Fabric 中存在和其他区块链系统不同的技术特性。下面我们将介绍这些技术特性。



说明

关于区块链 3.0 的定义目前社区还没有统一的结论，本书作者是从技术的框架和项目实现的角度认为 Fabric 属于区块链 3.0 范畴之内。这仅代表本书作者和相关技术社区的观点。

#### 4.1.1 Fabric 的多账本特性

在 Fabric 之前的区块链平台比如比特币或者以太坊，一般都只有一个账本，所有的记

录都在一个账本中。这样导致这个账本非常大，比如现在比特币的账本已经有 160G 左右。这种设计在比特币这样的公有链中是没有问题的，但是在 Fabric 这样的公有链中则是有问题。因此 Fabric 采用了多账本的设计方式。

在 Fabric 中有一个被称为通道 (channel) 的概念。通道本质是一个账本的逻辑概率。一个通道包含若干成员，这些成员之间共享同一个账本。通道内所有成员共享账本数据并且共同维护账本。一个通道可以包含多个会员，一个会员也可以在权限允许的情况下加入多个通道。同时不同的通道中账本数据的格式也是不一样的，Fabric 中账本的存储方式被设计成插件的形式，账本的数据可以选多种存储格式。不同的会员可以根据自己的实际情况选择不同的数据存储方式。

Fabric 的账本有以下特点：

- 使用基于 Key 的查询、范围查询、复合键查询来查询或更新账本。
- 只读查询支持丰富的查询语句 (CouchDB)。
- 只读的历史查询——实现数据追溯场景。
- 交易包含读取链码键值对 (读集)，以及写入链码键值对 (写集) 的版本。
- 交易包含所有背书节点提交至排序服务的签名。
- 交易被打包排序成区块，并通过通道从共识节点传至对等节点。
- 对等节点通过背书政策标准来验证交易。
- 在增加区块前，需要执行版本检查以确保数据在链码执行时间段没有被篡改。
- 当交易被验证并承诺后，便不可改变。
- 每个通道的账本都包含配置区块，它定义了政策标准、访问控制列表与其他相关信息。
- 通道包含了会员服务提供商实例，因此加密证书能传递到不同的证书颁发机构。

Fabric 的账本结构图如图 4-1 所示。

#### 4.1.2 Fabric 的智能合约

Fabric 中的智能合约称为链码 (英文名为: Chaincode, 链码英文名的中文直译, 在本书后面的章节中统称为 Chaincode), Chaincode 是一段用计算机语言编写的程序。Chaincode 运行在容器中, Fabric 通过 Chaincode 可以读取和修改账本数据, 同时会把交易的日志保存在状态数据库中。Chaincode 可以通过多种编程语言来开发, 目前支持 Go、Java、Node.js 等语言。但是目前支持最完善的还是 Go 语言版本的 Chaincode, 建议读者在实际的项目中使用 Go 语言来开发 Chaincode, 除非超级账本的官方发出其他语言版本的 Chaincode 已经成熟可用的信息。

Chaincode 是 Fabric 的重要组成部分, 在 Fabric 的技术架构中 Chaincode 的设计是非



常重要的一个环节, 在本书的第 7 章将重点介绍 Chaincode 的基本组成、技术特点和使用方法。

## Ledger

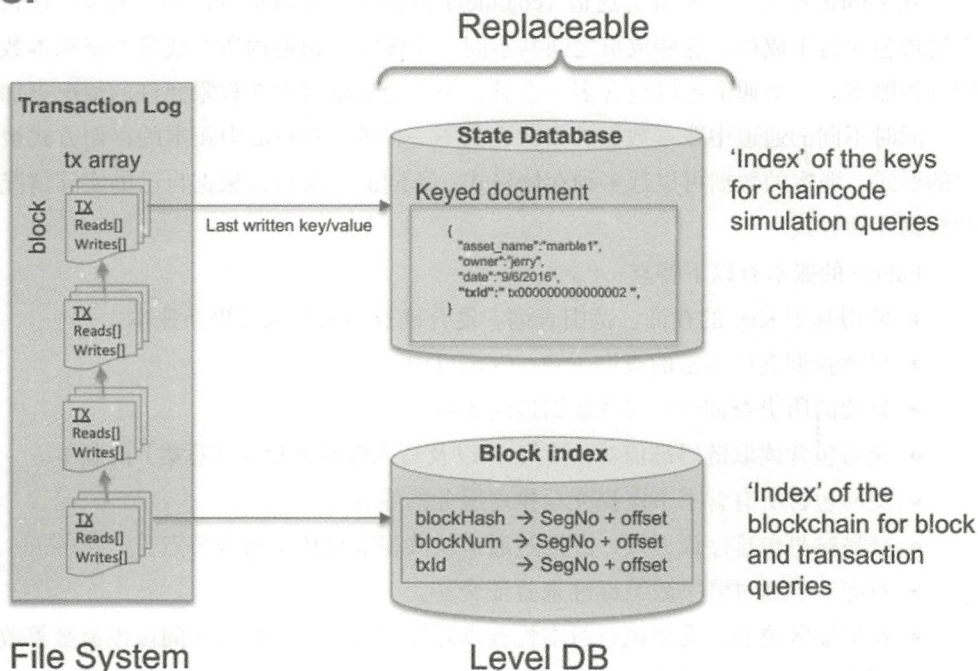


图 4-1 Fabric 账本结构图

### 4.1.3 Fabric 的权限系统

Fabric 和其他区块链的最大区别是: Fabric 的网络是不公开的, 如想进入网络中必须获取授权, 因此 Fabric 可以说是前面所提到的联盟链。通过前面的介绍我们知道类似比特币、以太坊等区块链是没有身份验证系统的, 任何人都可以在任何时间、任何能够接入网络的点加入区块链中, 这样的区块链称为公有链。在公有链区块链网络中任何人都可以加入网络中, 但是如想要获取交易的记账权需要付出代价 (算力), 这个代价就是我们经常提到的 POW (工作量证明算法), 通过这种方式使得单个节点的造假成本被无限扩大, 从而达到防止单个节点造假的目的。

在 Fabric 中没有采用类似 POW 这样的算法, 因此成员如果想加入网络必须获取授权, 否则不论算力多大都无法进入网络。为了解决成员授权加入的问题, Fabric 中有一个会员服务系统 (Membership Service Provider, MSP)。MSP 是基于 PKI 规范而建立的一个用户证书和私钥体系。MSP 是 Fabric 中非常重要的内容, 关于 Fabric 的 MSP 的详细内容我们将在本

书第6章详细介绍。

#### 4.1.4 Fabric 的共识算法

在 Fabric 中不同会员发起的交易是按照一定的顺序写入区块中，区块链之间连接起来之后形成区块链。在交易排序的过程需要防止恶意篡改，这不只在区块链系统中，在所有的分布式系统中都存在这个问题。在比特币中通过 POW（工作量证明）部分解决了这个问题。从 Fabric 提供的官方文档看，Fabric 会支持 Solo（单节点共识）、Kafka（分布式队列）和 SBFT（简单拜占庭容错）三种共识方式。但是在目前最新 Fabric 版本中只支持 Solo 和 Kafka 这两种共识算法，在未来的 Fabric 版本中会支持 PBFT 算法。

Solo 方法是指在单个节点中完成排序的方法，这种模式安全性和稳定性都比较差，如果单个节点出现问题，那么整个区块链系统都无法运行。因此 Solo 模式通常只是用在演示系统和本机开发环境中。

Kafka 是一种高吞吐量的分布式发布订阅消息系统。在 Fabric 的 Kafka 模式中，排序节点从 kafka 集群里获取相应 topic（kafka 的分区，用于在队列里隔离出多个数据域）的数据，以保证交易数据有序，这里借助了 kafka 的分布式一致机制实现对交易的排序。同时借助 Kafka，排序节点还可以进行集群，这样能有效地避免单点故障而导致整个网络崩溃的问题。

## 4.2 Hyperledger 中与 Fabric 相关的项目

Hyperledger 中除了 Fabric 项目外，一共有 13 个模块与 Fabric 相关。这些项目如下所示：

- Blockchain-explorer：Fabric 的管理工具，Blockchain-explorer 可以清楚查看 Fabric 的内部结构，如 channel、block、transactions、chaincode 等信息。Blockchain-explorer 是学习和使用 Fabric 的必备工具。
- Fabric-sdk-node：Node.js 语言版本的 Fabric SDK，基于 Node.js 实现了 Fabric 的 GRPC 通信协议。目前很多项目都基于 Fabric-sdk-node，使用 Fabric-sdk-node 需要熟悉掌握 Node.js 的语法结构。
- Fabric-sdk-java：Java 语言版本的 Fabric SDK，基于 Java 实现了 Fabric 的 GRPC 通信协议。如果对 Java 比较熟悉，可以使用 Fabric-sdk-java 进行应用程序的开发。
- Fabric-sdk-py：Python 语言版本的 Fabric SDK，基于 Python 实现了 Fabric 的 GRPC 通信协议。如果对 Python 比较熟悉，可以使用 Fabric-sdk-py 进行相关开发。
- Fabric-sdk-go：Go 语言版本的 Fabric SDK，基于 Go 实现了 Fabric 的 GRPC 通信协议。如果对 Go 比较熟悉，可以使用 Fabric-sdk-go 进行相关开发。
- Fabric-samples：Fabric 的例子，包含启动一个最简单的 Fabric 系统所必需的组件。

- Fabric-ca: Fabric-ca 是开源的 Fabric 证书服务器，基于 PKI 协议，提供了基于 JSON-RPC 协议的调用接口。Fabric-ca 包含 Fabric-ca-server 和 Fabric-ca-client 两个模块。Fabric-ca 是 Fabric 项目中不可或缺的部分，在本书的第 6 章中将详细介绍 Fabric-ca 的相关内容。
- Fabric-baseimage: Fabric 的 Docker 镜像文件的基础文件，其他的 Docker 镜像文件可以在此 Docker 镜像的基础上进行适当的修改。
- Fabric-chaincode-node: Nodejs 版本的 Chaincode，截至本书完稿时该项目处于验证阶段。
- Fabric-docs: Fabric 相关文档集合。
- Homebrew-fabric: 通过 brew 工具来安装 Fabric，在 MacOS 系统中比较适用。
- Fabric-sdk-rest: Fabric 的 REST 服务器，采用 Node.js 开发。
- Fabric-chaincode-java: Java 版本的 Chaincode，用 Java 语言来编写 Chaincode。该版本目前还在测试中。

以上项目的源代码都可以从 Hyperledger 在 Github 上面的项目中寻找并下载，Hyperledger 在 Github 中的项目组的地址如下所示：

<https://github.com/hyperledger>

### 4.3 Fabric 的模块、安装和使用

Fabric 的编译安装相对比较容易，读者在安装之前一定要根据自己的操作系统按照本书第 2 章的内容安装好相关的软件。下面的编译步骤适用于 Ubuntu、CentOS、MacOS 这三个平台。MacOS 由于系统的原因，某些地方可能需要单独设置，在需要设置的地方我们会有相关的提示。本节将通过一个完整的例子来让读者快速地了解 Fabric 中包含的相关模块和这些模块的编译步骤以及使用方法。

Fabric 并不是一个单独的程序而是由一组模块组成，这些模块中的每一个都是一个可独立运行的可执行文件。通过表 4-1 可以详细地了解这些模块的详细信息。

表 4-1 Fabric 的模块组成表

模块名称	功 能
peer	主节点模块，负责存储区块链数据，运行维护链码
orderer	负责对交易进行排序，并将排好序的交易打包成区块
cryptogen	组织和证书生成模块
configtxgen	区块和交易生成模块
configtxlator	区块和交易解析模块



这些模块在 Fabric 中都会起到非常重要的作用，开发一个 Fabric 应用都需要这些模块的参与。接下来我们会从编译和使用方式这两个方面来对这些模块有一个全面的介绍，在本书后面的内容中我们会通过一个简单的例子，让读者快速了解这些模块的作用，在本书的第5章中将详细介绍每个模块的功能和使用方法。

### 4.3.1 Fabric 的编译和安装

在使用 Fabric 的相关模块之前需要获取这些模块的二进制版本可执行文件。我们可以直接将 Fabric 源码编译成相关模块的二进制版本的可执行文件。在编译之前，请确保系统中已经成功安装了 Golang 的运行环境和相关的软件，具体可以参考本书 2.2.1 节的内容。Fabric 模块的编译过程如下：



**注意** Fabric 的最新版本支持的 Go 语言的版本是 1.9.x，请务必注意 Go 语言的版本。

第一步：创建目录并下载代码。

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric
```

第二步：安装相关依赖软件。

```
go get github.com/golang/protobuf/protoc-gen-go
mkdir -p $GOPATH/src/github.com/hyperledger/fabric/build/docker/gotools/bin
cp $GOPATH/bin/protoc-gen-go $GOPATH/src/github.com/hyperledger/fabric/build/
docker/gotools/bin
```



**注意** go get 之后编译好的文件会存放到环境变量 \$GOBIN 对应的目录中，如果没有设置 \$GOBIN 的值，系统默认将生成的文件存放到 \$GOPATH/bin 下面。

第三步：编译 Fabric 的模块。

进入 Fabric 源码所在的文件夹，执行以下命令可以一次完成 Fabric 5 个主要模块的编译过程，具体的命令如下所示：

```
cd $GOPATH/src/github.com/hyperledger/fabric
make release
make docker
```

对于 MacOS 系统，在编译之前需要进行以下设置：

- 打开文件 \$GOPATH/src/github.com/hyperledger/fabric/Makefile。

- 找到其中的第一个 GO\_LDFLAGS 字符串的位置, 在该字符串所在行的末尾加上字符串 -s。
- 保存文件 Makefile。

上述命令执行完成之后, 会自动将编译好的二进制文件存放在以下路径中:

- Ubuntu 和 CentOS 系统的存放路径:

```
$GOPATH/src/github.com/hyperledger/fabric/release/linux-amd64/bin
```

- MacOS 系统的存放路径:

```
$GOPATH/src/github.com/hyperledger/fabric/release/darwin-amd64/bin
```

第四步: Fabric 模块的安装。

编译完成之后, 这些模块就可以被运行了, 但是目前只能在编译文件所在的文件夹中运行这些模块, 这样是非常不方便的。为了更加方便地使用这些模块, 可以通过下面的命令将这些模块的可执行文件复制到系统目录中, 这样在系统中的任何路径下面都能运行这些可执行这些模块。

Ubuntu 和 CentOS 将 Fabric 模块编译后的文件复制到系统文件夹中的方法如下:

```
cp $GOPATH/src/github.com/hyperledger/fabric/release/linux-amd64/bin/* /usr/local/bin
```

MacOS 将 Fabric 模块编译后的文件复制到系统文件夹中的方法如下:

```
cp $GOPATH/src/github.com/hyperledger/fabric/release/darwin-amd64/bin/* /usr/local/bin
```

复制成功之后通过以下命令修改文件的执行权限, 否则无法执行。

```
sudo chmod -R 775 /usr/local/bin/configtxgen
sudo chmod -R 775 /usr/local/bin/configtxlator
sudo chmod -R 775 /usr/local/bin/cryptogen
sudo chmod -R 775 /usr/local/bin/peer
sudo chmod -R 775 /usr/local/bin/orderer
```

通过上面这些命令之后, 就可以在系统的任何路径下面运行这些模块了。

### 4.3.2 Fabric 模块安装结果检查

通过一组命令来检查安装过程是否成功。

执行命令 `peer version` 显示如下信息:

```
peer:
  Version: 1.1.0-snapshot-900850f
  Go version: go1.8.3
```

```

OS/Arch: linux/amd64
Experimental features: true
Chaincode:
  Base Image Version: 0.4.1
  Base Docker Namespace: hyperledger
  Base Docker Label: org.hyperledger.fabric
  Docker Namespace: hyperledger

```

执行命令 `orderer version` 显示如下信息：

```

orderer:
  Version: 1.1.0-snapshot-900850f
  Go version: go1.8.3
  OS/Arch: linux/amd64
  Experimental features: true

```

执行命令 `cryptogen version` 显示以下信息：

```

cryptogen:
  Version: development build
  Go version: go1.8.3
  OS/Arch: linux/amd64

```

执行命令 `configtxgen-version` 显示以下信息：

```

configtxgen:
  Version: development build
  Go version: go1.8.3
  OS/Arch: linux/amd64

```

执行命令 `configtxlator version` 显示以下信息：

```

configtxlator:
  Version: development build
  Go version: go1.8.3
  OS/Arch: linux/amd64

```

由于 Fabric 版本和运行操作系统的差异，这些命令执行完成之后的显示内容可能有一样的地方。但是只要能正确地显示版本信息且没有抛出异常，则表示这些 Fabric 的模块编译和安装是正确的，否则说明安装过程存在问题。请仔细检查上述编译步骤，直到所有检查命令都能显示正确的版本信息为止。如果没有正确安装 Fabric 的相关模块，下面的演示步骤是无法正确执行的。

### 4.3.3 利用 Docker 运行 Fabric 相关模块

使用 Fabric 的相关模块，除了前面介绍的直接编译源代码的方式，还可以通过 Docker 来运行这些模块。通过 Docker 运行 Fabric 相关模块有两种方法：通过本地源代码生成 Fabric 模块的 Docker 镜像文件和从 Docker 仓库中下载 Fabric 模块的镜像文件。下面将介绍



这两种方法的详细步骤。

### 1. 本地源代码生成 Fabric 模块的 Docker 镜像

通过本地源代码生成 Fabric 模块的 Docker 镜像文件是非常简单的，具体的操作命令如下所示：

```
cd $GOPATH/src/github.com/hyperledger/fabric
make docker
```

上述命令的执行过程涉及从 docker 的远程服务器中下载文件，因此需要等待一段时间，具体的安装时间视网络情况而定。在执行命令的过程中需要保持网络畅通。

### 2. 从 Docker 仓库中下载 Fabric 模块的 Docker 镜像文件

为了方便用户使用 Fabric，Fabric 开发小组会定期将 Fabric 稳定版本的源代码编译之后制作成 Docker 镜像文件，然后把这些 Docker 镜像文件上传到 Docker 仓库中供下载使用。从 Docker 仓库中下载 Fabric 模块 Docker 镜像文件的命令如下所示：

```
// 设置环境变量

export set ARCH=x86_64
export set BASEIMAGE_RELEASE=0.4.1
export set PROJECT_VERSION=1.1.0
export set IMG_TAG=1.1.0

// 获取 Docker 镜像

docker pull hyperledger/fabric-peer:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-orderer:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-ca:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-tool:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-ccenv:$ARCH-$PROJECT_VERSION
docker pull hyperledger/fabric-baseimage:$ARCH-$BASEIMAGE_RELEASE
docker pull hyperledger/fabric-baseos:$ARCH-$BASEIMAGE_RELEASE
```

上述命令也需要执行一段时间，执行完成之后可以通过命令 `docker ps` 查看镜像文件的列表，本例中的镜像文件如表 4-2 所示。

表 4-2 Fabric 的 Docker 镜像文件用途表

镜像文件名称	用 途
docker.io/hyperledger/fabric-baseos	基础镜像文件，其他镜像文件在该镜像文件的基础上生成
docker.io/hyperledger/fabric-baseimage	基础镜像文件包含了 jdk、golang、nodejs 等，可以用来生成 chaincode
docker.io/hyperledger/fabric-peer	peer 模块镜像文件
docker.io/hyperledger/fabric-ca	ca 模块镜像文件
hyperledger/fabric-tools	相关工具镜像文件，包含了 cryptogen、configtxgen、configtxlator 等工具

(续)

镜像文件名称	用 途
hyperledger/fabric-couchdb	couchdb 数据库镜像文件
hyperledger/fabric-kafka	kafka 库镜像文件
hyperledger/fabric-zookeeper	zookeeper 库镜像文件
hyperledger/fabric-testenv	测试环境库镜像文件
hyperledger/fabric-buildenv	编译环境库镜像文件
hyperledger/fabric-orderer	orderer 节点库镜像文件
hyperledger/fabric-javaenv	Java 版本的 chaincode 运行镜像文件
hyperledger/fabric-ccenv	Go 语言 chaincode 运行环境库镜像文件

在 Fabric 的源码中提供了一个基于 docker-compose 运行这些 Docker 镜像文件的例子，感兴趣的读者可以参考。示例代码的路径如下所示：

```
$GOPATH/src/github.com/hyperledger/fabric/examples/e2e_cli
```

Docker Compose 是在使用 Docker 容器部署分布式应用时的工具，可以定义哪个容器运行哪个应用。使用 Docker Compose，你只需定义一个多容器应用的 yml 文件，然后使用一条命令即可部署运行所有容器。关于 Docker Compose 的详细使用方法，限于篇幅本书无法详细讨论，有兴趣的读者可以参考相关的书籍。

## 4.4 快速运行一个简单的 Fabric 网络

Fabric 安装成功之后就可以启动相关的模块开始工作了，但是 Fabric 的配置和启动过程相对比较繁琐，在本书第 5 章会详细介绍各模块的使用方法。为了让读者快速地对 Fabric 有一个全面的认识，在本章中我们将演示如何快速搭建一个简单的 Fabric 网络，这个网络虽然简单但是包含 Fabric 核心的组成部分。

### 4.4.1 Fabric 环境准备

在开始运行下面的命令之前需要仔细检查运行机器的环境，必须满足以下两个条件：

- 按照第 2 章内容，安装好相关的工具。
- 按照 4.3.1 节的内容，顺利完成 Fabric 相关模块的编译步骤。

如果上述步骤出现问题，请自行检查相关步骤的过程，待所有步骤顺利完成之后方可继续运行下面的命令。



下面的演示我们将采用直接运行模块可执行文件的方式，不采用 Docker 的方式运行。

首先我们要选择一个目录来存在命令执行过程中生成的相关文件。这个目录可以是任意的位置,目录的名称符合操作系统的命名规则即可。具体的路径读者可以自行选择,这里我们给出一个测试的目录供大家参考。假设所有的配置文件和数据文件都存放在目录 `/opt/hyperledger` 中,执行如下命令:

```
mkdir -p /opt/hyperledger
```



**注意** 在测试系统中可以选择任意的位置,但是在生产系统中需要考虑数据的增长和数据的备份,应该让系统管理员提供空间充足的目录。

#### 4.4.2 生成 Fabric 需要的证书文件

启动 Fabric 之前首先需要生成相关的证书,生成证书是通过 `cryptogen` 模块完成的, `cryptogen` 模块会根据提供的配置文件生成后续模块运行过程中需要的证书和数据文件。在生成证书之前我们需要创建一个文件夹存放配置文件和生成的证书文件。本例中我们将配置文件和生成的证书文件放在文件夹 `/opt/hyperledger/fabricconfig` 中。

创建存放证书的文件夹的命令如下所示:

```
mkdir -p /opt/hyperledger/fabricconfig
```

`cryptogen` 提供了一个命令可以获取 `cryptogen` 模块所需要的配置文件的样式,该命令如下所示:

```
cryptogen showtemplate
```

把上述命令生成的内容复制到一个文件中稍加修改即可使用。本例中我们所使用的配置文件的内容如下所示:

```
OrdererOrgs:
  - Name: Orderer
    Domain: qklszzn.com
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: Org1
    Domain: org1.qklszzn.com
    Template:
      Count: 2
    Users:
      Count: 3
  - Name: Org2
    Domain: org2.qklszzn.com
    Template:
```



```
Count: 2
Users:
Count: 2
```



注意 在上面的配置文件中有一个属性 **Domain**，这个值是由用户设置的，在测试环境中这个值可随便设置，但生产环境中建议采用真实存在并且已经备案的域名。

将上述文件的内容保存到文件夹 `/opt/hyperledger/fabricconfig` 中，配置文件夹命名为 `crypto-config.yaml`。保存之后执行如下命令：

```
cd /opt/hyperledger/fabricconfig
cryptogen generate --config=crypto-config.yaml --output ./crypto-config
```

然后我们会发现在文件夹 `/opt/hyperledger/fabricconfig` 中会新增加一个文件夹 `crypto-config`，里面存放着本例的相关配置文件，可以通过 `tree` 命令查看生成证书文件的内容。

执行命令 `tree-L 5` 显示结果如下：

```
├── crypto-config
│   ├── ordererOrganizations
│   │   └── qklszzn.com
│   │       ├── ca
│   │       ├── msp
│   │       ├── orderers
│   │       ├── tlsca
│   │       └── users
│   └── peerOrganizations
│       ├── org1.qklszzn.com
│       │   ├── ca
│       │   ├── msp
│       │   ├── peers
│       │   ├── tlsca
│       │   └── users
│       └── org2.qklszzn.com
│           ├── ca
│           ├── msp
│           ├── peers
│           ├── tlsca
│           └── users
```



注意 `cryptogen` 生成的证书文件都有特别的作用，本章中我们暂不详细讨论，在本书第5章中会详细介绍这些文件的作用。这里只是让读者对 **Fabric** 有一个大概的认识，如果读者对 **Fabric** 的结构比较熟悉，想了解这些配置文件的作用，可以跳过本章后续内容直接阅读第5章。

通过上述步骤所有的证书文件都已经生成完毕, 现在需要将测试域名映射到本机的 IP 地址上面, 否则后面的操作可能会出现错误。

执行以下命令以便提取相关的域名:

```
cd /opt/hyperledger/fabricconfig
tree -L 5
```

在上述命令显示的内容中提取出后缀为 `qklszzn.com` 的域名。本例中提取出的信息如下:

```
orderer.qklszzn.com
peer0.org1.qklszzn.com
peer1.org1.qklszzn.com
peer3.org1.qklszzn.com
peer0.org2.qklszzn.com
peer1.org2.qklszzn.com
```

打开端映射文件:

```
vi /etc/hosts
```

在打开的文件中设置如下内容:

```
192.168.23.212 orderer.qklszzn.com
192.168.23.212 peer0.org1.qklszzn.com
192.168.23.212 peer1.org1.qklszzn.com
192.168.23.212 peer3.org1.qklszzn.com
192.168.23.212 peer0.org2.qklszzn.com
192.168.23.212 peer1.org2.qklszzn.com
```

输入以上内容之后保存 `/etc/hosts` 文件, 然后用 `ping` 命令测试以上配置是否正确。



**注意** 192.168.23.212 为本机的 IP 地址, 读者如果需要演示请根据自己的系统环境设置。

### 4.4.3 创始块的生成

#### 1. 系统创始块的生成

Fabric 是基于区块链的分布式账本, 每个账本都拥有自己的区块链, 账本的区块链中会存储账本的交易数据, 但账本区块链中的第一个区块是个例外, 该区块不存储交易数据而是存储配置信息, 通常将账本的第一个区块称为创始块。综上所述, Fabric 中账本的第一个区块是需要手动生成的。`configtxgen` 模块是专门负责生成系统的创始块和 Channel (Fabric 中的 Channel 就是账本, 关于 Channel 的概念在本书的后续章节会介绍) 的创始块。`configtxgen` 模块也需要一个配置文件来定义相关的属性。

下面是在 Fabric 源码中提供的 `configtxgen` 模块所需要的配置文件的例子。该文件的路

径是 \$GOPATH/src/github.com/hyperledger/fabric/sampleconfig，在这个目录下面有一个名为 configtx.yaml 的文件，对这个文件进行修改即可使用。由于初始块文件是提供给 Orderer 节点使用，因此我们创建一个文件夹来存储 Orderer 节点相关的文件。创建之后再把样例配置文件复制到该文件夹中。

创建存放 configtxgen 模块相关配置文件的文件夹的命令如下所示：

```
mkdir -p /opt/hyperledger/order/
cp -r $GOPATH/src/github.com/hyperledger/fabric/sampleconfig/configtx.yaml /
opt/hyperledger/order
cd /opt/hyperledger/order
```

对 configtx.yaml 进行修改，修改后的内容如下所示：

Profiles:

```
TestTwoOrgsOrdererGenesis:
  Orderer:
    <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
  Consortiums:
    SampleConsortium:
      Organizations:
        - *Org1
        - *Org2
```

```
TestTwoOrgsChannel:
  Consortium: SampleConsortium
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *Org1
      - *Org2
```

```
Organizations:
  - &OrdererOrg
    Name: OrdererOrg
    ID: OrdererMSP
    MSPDir: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/msp

  - &Org1
    Name: Org1MSP
    ID: Org1MSP
    MSPDir: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/msp
    AnchorPeers:
      - Host: peer0.org1.qklszzn.com
```



```

Port: 7051

- &Org2
  Name: Org2MSP
  ID: Org2MSP
  MSPDir: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org2.qklszzn.com/msp
  AnchorPeers:
    - Host: peer0.org2.qklszzn.com
      Port: 7051

Orderer: &OrdererDefaults

  OrdererType: solo
  Addresses:
    - orderer.qklszzn.com:7050
  BatchTimeout: 2s

  BatchSize:
    MaxMessageCount: 10
    AbsoluteMaxBytes: 98 MB
    PreferredMaxBytes: 512 KB

  Kafka:
    Brokers:
      - 127.0.0.1:9092
  Organizations:

Application: &ApplicationDefaults

  Organizations:

```

本章的主要目的是帮助读者快速了解 Fabric 的开发过程, 对 Fabric 的 5 个子模块的使用方法和相关参数没有进行详细说明, 在本书的第 5 章中会详细介绍这些模块的使用方法。

配置文件修改完成之后执行如下命令生成创始块文件。

```

cd /opt/hyperledger/order
configtxgen -profile TestTwoOrgsOrdererGenesis -outputBlock ./orderer.
genesis.block

```

上述命令执行完成之后会在文件夹 /opt/hyperledger/order 中生成文件 orderer.genesis.block。这是 Fabric 系统的创始块文件。

## 2. 账本创始块的生成

通道 (就是前面提到的账本, 在 Fabric 中称为 Channel, 本章后续内容中用 Channel 表示) 是 Fabric 中非常重要的概念, 一个 Channel 表示一个账本。Fabric 和其他区块链平

台最大的区别是 Fabric 支持多账本。每个 Fabric 应用都至少包含一个 Channel，因此创建 Channel 是 Fabric 中比较重要的步骤。本例我们将向读者演示如何创建一个 Channel。

与创建系统初始块的配置一样，创建 Channel 也是通过 configtxgen 模块完成的，在下面的例子中，Channel 的初始块的配置信息已经定义在前面生成的配置文件 configtx.yaml 中。

创建 Channel 的命令如下：

```
configtxgen -profile TestTwoOrgsChannel -outputCreateChannelTx ./roberttestchannel.tx -channelID roberttestchannel
```

上述命令执行完成之后会在目录生成文件 roberttestchannel.tx，该文件用来生成 Channel。除此之外还需要生成相关的锚点文件，而生成锚点文件需要执行以下命令：

```
configtxgen -profile TestTwoOrgsChannel -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID roberttestchannel -asOrg Org1MSP
```

```
configtxgen -profile TestTwoOrgsChannel -outputAnchorPeersUpdate ./Org2MSPanchors.tx -channelID roberttestchannel -asOrg Org2MSP
```

命令执行完成之后会在相应的文件夹下面生成文件 Org1MSPanchors.tx 和 Org2MSPanchors.tx，这些文件在后面会用到。

#### 4.4.4 Orderer 节点的启动

Orderer 节点负责交易的打包和区块的生成。Orderer 节点的配置信息通常放在环境变量或者配置文件中，本例中的配置信息统一存放在配置文件中。Fabric 源码提供了 Orderer 启动所用到的配置文件的实例，将示例配置文件复制到 Orderer 的文件夹下面稍加修改即可使用。

复制配置文件到 Orderer 文件夹的命令如下所示：

```
cd /opt/hyperledger/peer
cp $GOPATH/src/github.com/hyperledger/fabric/sampleconfig/orderer.yaml /opt/hyperledger/order
```

修改模板配置文件，由于篇幅有限，本例中我们只列出需要修改的部分。修改后配置文件中发生变化的内容如下：

General:

```
LedgerType: file
ListenAddress: 0.0.0.0
ListenPort: 7050
TLS:
  Enabled: false
  PrivateKey: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
```

```

qklszzn.com/orderers/orderer.qklszzn.com/tls/server.key
    Certificate: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/server.crt
    RootCAs:
        - /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/ca.crt
    ClientAuthEnabled: false
    ClientRootCAs:
    LogLevel: debug
    LogFormat: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
    GenesisMethod: file
    GenesisProfile: TestOrgsOrdererGenesis
    GenesisFile: /opt/hyperledger/order/orderer.genesis.block
    LocalMSPDir: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/msp
    LocalMSPID: OrdererMSP
    Profile:
        Enabled: false
        Address: 0.0.0.0:6060
    BCCSP:
        Default: SW
        SW:
            Hash: SHA2
            Security: 256
            FileKeyStore:
            KeyStore:
    FileLedger:
        Location: /opt/hyperledger/order/production/orderer
        Prefix: hyperledger-fabric-ordererledger
    RAMLedger:
        HistorySize: 1000
    Debug:
        BroadcastTraceDir:
        DeliverTraceDir:

```

要注意配置文件中的相关路径。

在配置文件 `orderer.yaml` 所在的目录执行如下命令启动 `orderer`:

```
orderer start
```

#### 4.4.5 Peer 节点的启动

Peer 模块是 Fabric 的核心节点, 所有的交易数据经过 Orderer 排序打包之后由 Peer 模块存储在区块链中。所有的 Chaincode 也是由 Peer 模块打包并且激活的。Peer 模块的配置信息同样由环境变量和配置文件组成, 本例中我们采用配置文件的方式来配置 peer 节点的参数。在设定配置文件之前需要创建一个文件夹存放 Peer 模块的配置文件和区块数据。在



Fabric 源码中同样提供了 Peer 模块配置文件的示例，将示例配置文件复制到 Peer 模块的文件夹下面稍加修改即可使用。

创建存储 Peer 模块的配置文件和区块数据的文件夹，并复制示例配置文件的命令如下所示：

```
mkdir -p /opt/hyperledger/peer
cd /opt/hyperledger/peer
cp $GOPATH/src/github.com/hyperledger/fabric/sampleconfig/core.yaml /opt/hyperledger/peer
```

修改后 Peer 模块配置文件中变化的内容如下所示：

```
logging:
  peer:      debug
  cauthdsl:  warning
  gossip:    warning
  ledger:    info
  msp:       warning
  policies:  warning
  grpc:      error
  format:    '%(color){time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
  peer:

  id: peer0.org1.qklszzn.com
  networkId: dev
  listenAddress: 0.0.0.0:7051
  chaincodeListenAddress: 0.0.0.0:7052
  address: peer0.org1.qklszzn.com:7051
  addressAutoDetect: false
  gomaxprocs: -1
  gossip:

    bootstrap: 127.0.0.1:7051
    useLeaderElection: true
    orgLeader: false
    endpoint:
    maxBlockCountToStore: 100
    maxPropagationBurstLatency: 10ms
    maxPropagationBurstSize: 10
    propagateIterations: 1
    propagatePeerNum: 3
    pullInterval: 4s
    pullPeerNum: 3
    requestStateInfoInterval: 4s
    publishStateInfoInterval: 4s
    stateInfoRetentionInterval:
    publishCertPeriod: 10s
    skipBlockVerification: false
```

```

dialTimeout: 3s
connTimeout: 2s
recvBuffSize: 20
sendBuffSize: 200
digestWaitTime: 1s
requestWaitTime: 1s
responseWaitTime: 2s
aliveTimeInterval: 5s
aliveExpirationTimeout: 25s
reconnectInterval: 2
externalEndpoint: peer0.org1.qklszn.com:7051
election:
  startupGracePeriod: 15s
  membershipSampleInterval: 1s
  leaderAliveThreshold: 10s
  leaderElectionDuration: 5s
pvtData:
  maxPeers: 3
  minAck: 3
events:
  address: 0.0.0.0:7053
  buffersize: 100
  timeout: 10ms
tls:
  enabled: false
  cert:
    file: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszn.com/peers/peer0.org1.qklszn.com/tls/server.crt
  key:
    file: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszn.com/peers/peer0.org1.qklszn.com/tls/server.key
  rootcert:
    file: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszn.com/peers/peer0.org1.qklszn.com/tls/ca.crt
  serverhostoverride:
fileSystemPath: /opt/hyperledger/peer/production
BCCSP:
  Default: SW
  SW:
    Hash: SHA2
    Security: 256
    FileKeyStore:
      KeyStore:

mspConfigPath: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszn.com/peers/peer0.org1.qklszn.com/msp

localMspId: Org1MSP
profile:
  enabled: false
  listenAddress: 0.0.0.0:6060

```

```

handlers:
  authFilter: "DefaultAuth"
  decorator: "DefaultDecorator"
vm:
  endpoint: unix:///var/run/docker.sock
  docker:
    tls:
      enabled: false
      ca:
        file: docker/ca.crt
      cert:
        file: docker/tls.crt
      key:
        file: docker/tls.key

  attachStdout: false
  hostConfig:
    NetworkMode: host
    Dns:
    LogConfig:
      Type: json-file
      Config:
        max-size: "50m"
        max-file: "5"
    Memory: 2147483648
chaincode:
  peerAddress:
  id:
    path:
    name:
  builder: $(DOCKER_NS)/fabric-ccenv:${ARCH}-${PROJECT_VERSION}
  golang:
    runtime: $(BASE_DOCKER_NS)/fabric-baseos:${ARCH}-${BASE_VERSION}
  car:
    runtime: $(BASE_DOCKER_NS)/fabric-baseos:${ARCH}-${BASE_VERSION}
  java:
    Dockerfile: |
      from $(DOCKER_NS)/fabric-javaenv:${ARCH}-${PROJECT_VERSION}
  node:
    runtime: $(BASE_DOCKER_NS)/fabric-baseimage:${ARCH}-${BASE_VERSION}
  startupTimeout: 300s
  execTetetimeout: 30s
  mode: dev
  keepalive: 0
  system:
    csc: enable
    lsc: enable
    esc: enable
    vsc: enable
    qsc: enable
    rsc: disable

```



```

logging:
  level: info
  shim: warning
  format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
ledger:

blockchain:
state:
  stateDatabase: goleveldb
  couchDBConfig:
    couchDBAddress: 127.0.0.1:5984
    username:
    password:
    maxRetries: 3
    maxRetriesOnStartup: 10
    requestTimeout: 35s
    queryLimit: 10000
history:
  enableHistoryDatabase: true

```

在配置文件 `core.yaml` 所在的文件夹中执行以下命令启动 `order` 节点。

```

export set FABRIC_CFG_PATH=/opt/hyperledger/peer
peer node start >> log_peer.log 2>&1 &

```

#### 4.4.6 创建通道

现在我们可以创建通道，创建通道的过程一共分为三个步骤。

第一步：创建通道。

```

export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer channel create -t 50 -o orderer.qklszzn.com:7050 -c roberttestchannel -f
/opt/hyperledger/order/roberttestchannel.tx

```

创建通道完成之后，会在执行命令的当前目录生成名为“`roberttestchannel.block`”的通道初始块文件。

第二步：让已经运行的 `Peer` 模块加入通道。

```

export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer channel join -b /opt/hyperledger/order/roberttestchannel.block

```

在上述创建通道的命令中，“`-b`”后面的参数为第一步生成的文件“`roberttestchannel`”。

block”，需要注意这个文件的路径。

第三步：更新锚节点。

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp
```

```
peer channel update -o orderer.qklszn.com:7050 -c roberttestchannel -f /opt/
hyperledger/order/Org1MSPanchors.tx
```

#### 4.4.7 Chaincode 的部署和调用

现在可以部署一个 Chaincode（关于 Chaincode 的详细内容在本书第 7 章会有详细的介绍）来测试 Peer 节点和 Orderer 节点的部署是否正确。这里采用 Fabric 源码自带的例子来作为测试 Chaincode。测试 Chaincode 的源代码路径如下所示：

```
$GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
example02
```

Chaincode 相关的测试一共有四个步骤。

第一步：部署 Chaincode 代码。

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp
```

```
peer chaincode install -n r_test_cc6 -v 1.0 -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02
```

第二步：实例化 Chaincode 代码。

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp
```

```
peer chaincode instantiate -o om:7050 -C roberttestchannel -n r_test_cc6 -v 1.0
-c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

第三步：通过 Chaincode 写入数据。

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp
```

```
peer chaincode invoke -o orderer.qklszzn.com:7050 -C roberttestchannel -n r_test_cc6 -c '{"Args":["invoke","a","b","1"]}'
```

第四步：通过 Chaincode 查询数据。

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

```
peer chaincode query -C roberttestchannel -n r_test_cc6 -c '{"Args":["query","a"]}'
```

如果上述命令都能正确执行，那么一个简单的 Fabric 系统就已经部署完成了。

## 4.5 本章小结

本章主要介绍了运行一个 Fabric 系统所需要的步骤，包括 Fabric 核心模块的编译、运行和配置。同时还介绍了 Fabric 核心概念 Channel 的创建和使用，最后介绍了 Fabric 的智能合约——Chaincode 的安装、实例化以及调用的方法。熟悉本章的内容有助于读者快速了解 Fabric 系统的组成部分和开发流程，为阅读本书后续章节打下基础。





## Fabric 核心模块详解

本章主要介绍 Fabric 的核心模块以及这些核心模块配置信息的管理方式。本章内容是第 4 章内容的延续和补充，在第 4 章没有详细说明的内容，比如各个模块的命令选项和配置选项等信息，在本章会有比较详细的说明。通过本章内容读者可以对 Fabric 核心模块的命令选择和配置方式有更加详细的了解，为后续内容的阅读打下坚实的基础。

### 5.1 Fabric 的核心模块功能、通用选项和命令

#### 5.1.1 Fabric 核心模块及其功能

前面我们知道 Fabric 是一个由 5 个核心模块组成的程序组。在 Fabric 被成功编译完成之后，一共有 5 个核心模块。这 5 个模块如表 5-1 所示。

表 5-1 Fabric 的 5 大核心模块及其功能

模块名称	功 能
peer	主节点模块，负责存储区块链数据，运行维护链码
orderer	交易打包、排序模块
cryptogen	组织和证书生成模块
configtxgen	区块和交易生成模块
configtxlator	区块和交易解析模块

这 5 个模块中 peer 和 orderer 属于系统模块，cryptogen、configtxgen 和 configtxlator 属

于工具模块。工具模块负责证书文件、区块链创始块、通道创始块等相关文件和证书的生成工作,但是工具模块不参与系统的运行。peer 模块和 orderer 模块作为系统模块是 Fabric 的核心模块,启动之后会以守护进程的方式在系统后台长期运行。

### 5.1.2 Fabric 模块的通用选项和命令

Fabric 的 5 个核心模块都是基于命令行的方式运行的,目前 Fabric 没有为这些模块提供相关的图形界面,因此想要熟练使用 Fabric 的这些核心模块,必须熟悉这些模块的命令选项。在 Fabric 核心模块的命令选项中有一些通用的选项(所有的 Fabric 核心模块都具备这些通用选项),下面我们介绍一下这些通用选项。

#### 1. --help 选项

help 选项将显示该命令行模块的所有选项。help 选项的内容通常分为三部分:

- Available Commands: 表示表命令,包含子命令。
- Flags: 显示当前命令的参数。
- Global Flags: 表示全局参数,所有的子命令都可以使用。

help 还可以查询子命令的选项信息,具体的命令格式如下所示:

```
模块名 + 子命令 + --help
```

上述命令将显示子命令的子命令、子命令的选型及全局选项。

#### 2. -v, --version

-v 和 --version 表示的含义是相同的,都表示获取当前模块的版本信息。

--version 选项虽然简单,但是用途比较广泛,特别是在使用基于 Fabric 的第三方应用,需要注意其适配的 Fabric 的版本时,--version 选项就非常有用。

## 5.2 Fabric 模块的子命令、选项和配置文件

Fabric 核心模块的配置信息由配置文件、命令行选型、环境变量这三个部分组成。这也是初学者经常容易混淆的地方,特别是配置文件和环境变量之间的关系。尤其是 peer 和 orderer 这两个模块,经常会因为环境变量和配置文件的关系导致系统启动失败。

我们举个例子,peer 模块有个全局选项 --logging-level,这个选项会设定 peer 模块启动后系统日志的级别,同时在配置文件 logging 节点下面的 peer 子节点也可以设定 peer 模块启动后系统日志的级别,此外环境变量 CORE\_LOGGING\_LEVEL 也可以设定 peer 模块启动后系统日志的级别。

那么问题来了,如果上述三种参数配置方式同时出现,peer 模块或者 orderer 模块会听

谁的呢？经过分析 Fabric 的源代码，我们发现上述三种配置方式之间存在下面关系。

环境变量 > 配置文件 > 命令选项

环境变量和配置文件是可以相互转化的，我们会在后面详细介绍。这里读者可能会有一个疑问，为什么已经有配置文件还需要引入环境变量呢？这个问题 Fabric 的官方似乎没有给出答案，我们经过分析认为可能是为了适应 Docker 容器而加入了环境变量吧。



**注意** 环境变量和配置文件可以相互转化，但是我们还是建议读者在运行 Fabric 的系统的时候尽量把参数配置在统一的格式中，要么全部配置在环境变量中，要么全部配置在配置文件中。如果是基于 Docker 运行的，建议采用环境变量的配置方式。如果是用命令直接启动，建议采用配置文件参数配置方式。

下面将详细介绍 Fabric 的 5 个核心模块的命令选项、配置文件、环境变量等内容。

### 5.2.1 cryptogen

cryptogen 模块主要用来生成组织结构和账号相关的文件，任何 Fabric 系统的开发通常都是从 cryptogen 模块开始的。在 Fabric 项目中，当系统设计完成之后第一项工作就是根据系统的设计来编写 cryptogen 的配置文件，然后通过这些配置文件生成相关的证书文件。cryptogen 模块所使用的配置文件是整个 Fabric 项目的基石。下面我们将介绍 cryptogen 模块命令行选项及其使用方式。

#### 1. cryptogen 模块命令说明

cryptogen 模块是通过命令行的方式运行的，一个 cryptogen 命令由命令行参数和配置文件两部分组成，通过执行命令 `cryptogen --help` 可以显示 cryptogen 模块的命令行选项，执行结果如下所示：

```
usage: cryptogen [<flags>] <command> [<args> ...]
Utility for generating Hyperledger Fabric key material
Flags:
  --help
Commands:
  help
  generate
  showtemplate
  version
```

cryptogen 模块一共有 4 个命令，这 4 个命令及其作用如下所示：

- `help`：显示帮助信息。
- `generate`：根据配置文件生成证书信息。



- `showtemplate`: 显示系统默认 `cryptogen` 模块配置文件信息。
- `version`: 显示当前模块的版本号。

其中 `generate` 命令选项是用来根据配置文件生成 Fabric 系统相关的证书文件。

## 2. cryptogen 模块的配置文件

`cryptogen` 模块的配置文件用来描述需要生成的证书文件的特性, 比如: 有多少个组织, 有多少个节点, 需要多少个账号等。这里我们通过一个 `cryptogen` 模块配置文件的具体例子来初步了解配置文件的结构, 该例子是 Fabric 源代码中自带的示例, 其详细路径如下所示:

[https://github.com/hyperledger/fabric/blob/release/examples/e2e\\_cli/cryptogen-config.yaml](https://github.com/hyperledger/fabric/blob/release/examples/e2e_cli/cryptogen-config.yaml)

该示例的内容和相关节点的注释如下所示:

```
OrdererOrgs:                                // 定义 orderer 节点
- Name: Orderer                             // orderer 节点的名称
  Domain: example.com                      // orderer 节点的根域名
  Specs:
    - Hostname: orderer                    // orderer 节点的主机名

PeerOrgs:
- Name: Org1                                // 组织 1 的名称
  Domain: org1.example.com                 // 组织 1 的根域名
  Template:
    Count: 2                               // 组织 1 中的节点数目
  Users:
    Count: 1                               // 组织 1 中的用户数目
- Name: Org2
  Domain: org2.example.com
  Template:
    Count: 2
  Users:
    Count: 1
```

上述模板文件定义了一个 `orderer` 节点, 这个 `orderer` 节点的名字为 `orderer`, `orderer` 节点的根域名为 `example.com`, 主机名为 `orderer`。模板文件同时定义了两个组织, 两个组织的名字分别为 `org1` 和 `org2`, 其中组织 `org1` 包含了 2 个节点和 1 个用户, 组织 `org2` 包含 2 个节点和 1 个用户。

除了 Fabric 源码中提供的例子, 还可以通过命令 `cryptogen showtemplate` 获取默认的模板文件, 在实际项目中稍加修改这些默认的模板文件即可使用。

## 3. cryptogen 实例: 创建测试配置文件

- 创建测试配置文件

现在我们通过一个例子来具体演示 `cryptogen` 模块的使用。我们将定义一个测试用

的 Fabric 系统，首先给整个系统定义一个根域名 qklszzn.com。orderer 节点我们命名为 Orderer，在该测试系统中我们假设有三个组织，分别命名为 org1, org2, org3。其中组织 org1 包含 4 个节点和 6 个用户，组织 org2 包含 5 个节点和 11 个用户，组织 org3 包含 3 个节点和 13 个用户。我们将测试用 Fabric 系统的相关信息如表 5-2 和表 5-3 所示。

表 5-2 cryptogen 模块测试用 Fabric 系统基本信息表（一）

属性名称	属性值
系统根域名	qklszzn.com
系统 orderer 节点名称	Orderer

表 5-3 cryptogen 模块测试用 Fabric 系统组织信息表（二）

组成名称	peer 节点数	用户数
org1	4	6
org2	5	11
org3	3	13

根据上述 Fabric 系统的基本信息，我们可以编写 cryptogen 模块用的配置文件，配置文件的内容如下所示：

```
OrdererOrgs:
  - Name: Orderer
    Domain: qklszzn.com
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: Org1
    Domain: org1.qklszzn.com
    Template:
      Count: 4
    Users:
      Count: 6
  - Name: Org2
    Domain: org2.qklszzn.com
    Template:
      Count: 5
    Users:
      Count: 11
  - Name: Org3
    Domain: org3.qklszzn.com
    Template:
      Count: 3
    Users:
      Count: 13
```

在测试环境中域名可以随便定义的，但是在正式的生产环境中，域名最好选择已经通

过备案的, 如果域名没有备案, 需要让机房的防火墙把没有备案的域名放到白名单中。

- 生成证书文件

我们通过 `cryptogen` 模块的 `generate` 命令可以生成相关的证书文件。命令如下所示:

```
cryptogen generate --config=/opt/hyperledger/fabricconfig/crypto-config.yaml
--output /opt/hyperledger/fabricconfig/crypto-config
```

`/opt/hyperledger/fabricconfig` 是证书文件存放目录, 也可以是任何具有读写权限的文件夹, 但是需要提前创建。

进入 `/opt/hyperledger/fabricconfig/crypto-config` 文件夹之后有两个子文件夹, 通过命令 `tree -L 2` 显示如下:

```
├── ordererOrganizations    // orderer 节点相关的证书文件
└── peerOrganizations      // 组织相关的证书文件, 包括组织的节点数、用户数等证书文件
```

`tree` 工具是 Linux 系统中的常用命令, 可以显示文件夹中文件的层次结构, 在管理和维护类 Linux 系统的时候非常方便。关于 `tree` 工具的详细介绍, 可以参考本书的第 2 章。

命令执行成功之后, 进入 `ordererOrganizations` 子文件夹, 然后通过命令 `tree -L 4` 显示如下:

```
├── qklszzn.com             // 根域名为 qklszzn.com 的 orderer 节点的相关证书文件
│   ├── ca                 // CA 服务器签名文件
│   │   ├── 12df544014743f9f6c243807428d56ea316dd0923bcff30429ed61d2aa8a2bd0_sk
│   │   └── ca.qklszzn.com-cert.pem
│   ├── msp
│   │   ├── admincerts    // orderer 管理员的证书
│   │   │   └── Admin@qklszzn.com-cert.pem
│   │   ├── cacerts      // orderer 根域名服务器的签名证书
│   │   │   └── ca.qklszzn.com-cert.pem
│   │   ├── tlscacerts   // TLS 连接用的身份证书
│   │   │   └── tlsca.qklszzn.com-cert.pem
│   ├── orderers         // orderer 节点需要的相关证书文件
│   │   └── orderer.qklszzn.com
│   │       ├── msp      // orderer 节点相关证书
│   │       └── tls      // orderer 节点和其他节点 TLS 连接用的身份证书
│   ├── tlsca
│   │   ├── da3ef4d128668ea83219084777a4edd3e884b80674718da5b7047fb082f15175_sk
│   │   └── tlsca.qklszzn.com-cert.pem
│   └── users             // orderer 节点用户相关的证书
│       └── Admin@qklszzn.com
│           ├── msp
│           └── tls
```

在实际的开发中 `orderer` 节点的这些证书其实不需要直接使用, 只是在 `orderer`



节点启动时指明项目的位置即可，在下文关于 orderer 节点的配置环节中将详细描述如何配置这些证书文件。

#### 4. Fabric 证书文件的结构

cryptogen 模块生成的证书文件就是 Fabric 系统运行所需要的证书文件，接下来我们将详细介绍这些证书文件的种类和作用。进入文件夹 peerOrganizations 后执行命令 `tree -L 1`，命令结果如下所示：

```
├── org1.qklszzn.com
├── org2.qklszzn.com
└── org3.qklszzn.com
```

通过上述命令结果我们可以发现，在文件夹 peerOrganizations 中包含三个子文件，从这三个子文件夹的命名可以发现它们分别对应前面配置文件中定义的组织（可以参考表 5-3 的相关内容）。文件夹 peerOrganizations 中的三个子文件夹和表 5-3 中描述的组织对应关系如表 5-4 所示。

表 5-4 cryptogen 模块生成的证书文件和配置文件定义的组织对应表

组成名称	证书文件夹名称
org1	org1.qklszzn.com
org2	org2.qklszzn.com
org3	org3.qklszzn.com

由于存放这三个组织的文件夹的结构都是一样的，下面我们以组织 org1 为例来说明这些文件的作用。由于 org1 的用户和节点比较多，但是相同的节点和用户的配置是一样，所以示例中我们只保留一个节点和一个用户的配置信息。



**注意** 上述的节点是指组中的 peer 节点。

进入文件夹 org1.qklszzn.com 执行命令 `tree -L 5` 显示如下：

```
├── ca // 根节点签名证书
│   ├── aldb721c0cfb6f107fc45501d29866633d21a3492c87bb352687b2e8e85b652e_sk
│   └── ca.org1.qklszzn.com-cert.pem
├── msp
│   ├── admincerts // 组织管理员的证书
│   │   └── Admin@org1.qklszzn.com-cert.pem
│   ├── cacerts // 组织的根证书
│   │   └── ca.org1.qklszzn.com-cert.pem
│   └── tlscacerts // TLS 连接身份证书
│       └── tlsca.org1.qklszzn.com-cert.pem
```



cryptogen 模块生成的证书是 Fabric 系统的重要组成部分, 是所有 Fabric 系统的开始, 在后面的案例中将详细地向读者介绍这些证书的作用。

## 5.2.2 configtxgen

### 1. configtxgen 模块的命令

configtxgen 模块用来生成 orderer 的初始化文件和 channel 的初始化文件。configtxgen 模块包含如下子命令选项:

- asOrg: 所属的组织。
- channelID: channel 名字, 如果没有系统会提供一个默认值。
- inspectBlock: 打印制定区块文件中的配置内容。
- inspectChannelCreateTx: 打印创建通道的交易的配置文件。
- outputAnchorPeersUpdate: 更新 channel 配置信息。
- outputBlock: 输出区块文件的路径。
- outputCreateChannelTx: 标示输出创始块文件。
- profile: 配置文件的节点。

- version: 显示版本信息。

## 2. configtxgen 模块的配置文件

configtxgen 模块的配置文件包含 Fabric 系统初始块、Channel 初始块文件等信息。

configtxgen 模块配置文件的样例如下所示:

配置文件中有些注释内容比较多, 因此在配置文件的样例下面会有详细的说明。

Profiles:

```
// 以下部分定义了整个系统的配置信息
// 组织定义标示符, 可自定义, 命令中的 -profile 参数对应该标识符
```

```
TestOrgsOrdererGenesis:
    // orderer 配置属性, 系统关键字不得更改
    Orderer:

        // 引用下面名为 OrdererDefaults 的属性
        <<: *OrdererDefaults
        Organizations:
            // 引用下面的名为 OrdererOrg 的属性
            - *OrdererOrg
```

```
// 定义了系统中包含的组织
Consortiums:
    SampleConsortium:
```

```
        // 系统中包含的组织
        Organizations:
            - *Org1    // 引用了下文定义配置
            - *Org2
            - *Org3
```

```
// 以下内容是以 channel 的配置信息
// 通道定义标示符, 可自定义
```

```
TestOrgsChannel:
    Consortium: SampleConsortium
    Application:
        <<: *ApplicationDefaults
        Organizations:
            - *Org1
            - *Org2
            - *Org3
```

```
// orderer 节点相关信息
```



Organizations:

// orderer 节点配置信息

- &OrdererOrg

// orderer 节点名称

Name: OrdererOrg

// orderer 节点编号

ID: OrdererMSP

// msp 文件夹路径

MSPDir: msp

// Orderer 节点中包含的组织, 如果有的多个组织可以配置多个, 本例中一共有三个组织

- &Org1

Name: Org1MSP // 组织名称

ID: Org1MSP // 组织编号

MSPDir: msp // 组织 msp 文件名

AnchorPeers: // 组织的访问域名和端口

- Host: peer0.org1.qklszzn.com

Port: 7051

- &Org2

Name: Org2MSP

ID: Org2MSP

MSPDir: msp

AnchorPeers:

- Host: peer0.org2.qklszzn.com

Port: 7051

- &Org3

Name: Org3MSP

ID: Org3MSP

MSPDir: msp

AnchorPeers:

- Host: peer0.org2.qklszzn.com

Port: 7051

// orderer 节点的配置信息

Orderer: &OrdererDefaults

OrdererType: solo

// orderer 节点共识方法

Addresses:

- orderer.qklszzn.com:7050

// orderer 监听的地址

BatchTimeout: 2s

BatchSize:

MaxMessageCount: 10

AbsoluteMaxBytes: 98 MB

PreferredMaxBytes: 512 KB

```
// kafka 相关配置
Kafka:
  Brokers:
    - kafka0:9092
    - kafka1:9092
    - kafka2:9092
    - kafka3:9092
  Organizations:

Application: &ApplicationDefaults
Organizations:
```

**Profiles 节点详解：**Profiles 节点定义了整个系统的结构和 channel 的结构，配置文件中的 Profiles 关键字不允许更改，否则配置无效。系统配置信息中设置了系统中 orderer 节点的信息以及系统中包含的组织数。

### 3. configtxgen 典型的应用场景

#### (1) 创建 orderer 的初始块

创建 orderer 初始块的命令格式如下：

```
configtxgen -profile TestTwoOrgsOrdererGenesis -outputBlock ./orderer.
genesis.block
```

TestTwoOrgsOrdererGenesis 要和配置文件中的配置选项对应。可以由字母和数字组成，建议不要有特殊符号。

#### (2) 创建 channel 初始块

创建 channel 初始块的命令格式如下：

```
configtxgen -profile TestTwoOrgsChannel -outputCreateChannelTx ./
roberttestchannel.tx -channelID roberttestchannel
```

#### (3) 创建锚点更新文件

创建锚点更新文件的命令格式如下：

```
configtxgen -profile TestTwoOrgsChannel -outputAnchorPeersUpdate ./
Org1MSPanchors.tx -channelID roberttestchannel -asOrg Org1MSP
```

### 5.2.3 configtxlator

configtxlator 模块可以把区块链的二进制文件转化成 JSON 格式的文件，便于我们阅读和理解。在本节中我们将介绍命令的调用方式及参数，而 configtxlator 命令的使用方法会在后文中用一个完整的示例详细阐述。

configtxlator 模块包含三个命令，执行命令 configtxlator --help 如下所示：

```
Commands:
```

```

help      // 显示帮助信息
start     // 启动 configtxlator REST 服务器
version   // 显示版本信息

```

其中 start 命令包含两个参数:

```

--hostname // configtxlator REST 服务器绑定的 IP 地址
           // 0.0.0.0 表示绑定所有的网卡地址

--port     // configtxlator REST 服务器绑定的端口

```

configtxlator 是以一个 RESTAPI 服务的形式提供服务的, 可以通过标准的 HTTP 请求来访问 configtxlator。configtxlator 启动示例如下:

```
configtxlator start --hostname=0.0.0.0 --port=
```

configtxlator 的 REST 服务提供了解码、编码, 计算配置更新、交易打包四个功能。下面分别介绍这四个功能的调用方式。



**注意** 下面的测试中我们通过 curl 来发起请求, 在测试之前请确保系统中已经正确地安装了 curl 模块。

---

### 1. 解码

```
curl -X POST --data-binary @configuration_block.block http://127.0.0.1:7059/
protolator/decode/common.Block > ./configuration_block.json
```

上述命令把当前目录下的区块文件 configuration\_block.block 转换成名为 configuration\_block.json 的 JSON 格式的文件。

### 2. 编码

```
curl -X POST --data-binary @updated_config.json http://127.0.0.1:7059/
protolator/encode/common.Config > updated_config.pb
```

上述命令将 JSON 格式的配置文件 pdated\_config.json 转化成区块链文件 updated\_config.pb。

### 3. 计算配置更新量

```
curl -X POST -F original=@config.pb -F updated=@updated_config.
pb http://127.0.0.1:7059/configtxlator/compute/update-from-configs -F
channel=testchainid > config_update.pb
```

对比配置文件 config.pb 和 updated\_config.pb 的差异, 提取其中的差异并生成区块格式的文件 config\_update.pb。



## 4. 交易打包

```
curl -X POST --data-binary @config_update_as_envelope.json http://127.0.0.1:8188/protolator/encode/common.Envelope > config_update_as_envelope.tx
```

把 JSON 格式的交易文件 config\_update\_as\_envelope.json 打包成交易格式的文件 config\_update\_as\_envelope.tx。

在本书的第 10 章中将会详细介绍 configtxlator 模块的使用方法。

## 5.2.4 orderer

orderer 模块负责对交易进行排序，并将排好序的交易打包成区块。

### 1. orderer 模块的命令和参数

```
help    显示帮助信息
start*  启动 orderer 节点
version 显示版本信息
Show version information
benchmark 采用基准本模式运行 orderer
```

### 2. orderer 模块的配置信息

orderer 模块配置信息可以用环境变量或者配置文件的方式来配置，环境变量的配置示例如下所示：

```
export set ORDERER_GENERAL_LOGLEVEL=debug
export set ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
export set ORDERER_GENERAL_LISTENPORT=7050
export set ORDERER_GENERAL_GENESIMETHOD=file
export set ORDERER_GENERAL_GENESISFILE=/opt/hyperledger/order/orderer.genesis.
block
export set ORDERER_GENERAL_LOCALMSPID=OrdererMSP
export set ORDERER_GENERAL_LOCALMSPDIR=/opt/hyperledger/fabricconfig/crypto-config/
ordererOrganizations/qklszn.com/orderers/orderer.qklszn.com/msp
export set ORDERER_GENERAL_LEDGERTYPE=file
export set ORDERER_GENERAL_BATCHTIMEOUT=10s
export set ORDERER_GENERAL_MAXMESSAGECOUNT=10
export set ORDERER_GENERAL_TLS_ENABLED=false
export set ORDERER_GENERAL_TLS_PRIVATEKEY=/opt/hyperledger/fabricconfig/crypto-
config/ordererOrganizations/qklszn.com/ /orderer.qklszn.com/tls/server.key
export set ORDERER_GENERAL_TLS_CERTIFICATE=/opt/hyperledger/fabricconfig/
crypto-config/ordererOrganizations/qklszn.com/orderers/orderer.qklszn.com/tls/
server.crt
export set ORDERER_GENERAL_TLS_ROOTCAS=[/opt/hyperledger/fabricconfig/crypto-
config/ordererOrganizations/qklszn.com/orderers/orderer.qklszn.com/tls/ca.crt]
```

orderer 模块配置文件示例如下所示：

General:

```

    LedgerType: file
    ListenAddress: 0.0.0.0
    ListenPort: 7050

    TLS:
        Enabled: false
        PrivateKey: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/server.key
        Certificate: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/server.crt
        RootCAs:
            - /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/ca.crt
        ClientAuthEnabled: false
        ClientRootCAs:

    LogLevel: debug
    LogFormat: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
    GenesisMethod: file
    GenesisProfile: TestOrgsOrdererGenesis
    GenesisFile: /opt/hyperledger/order/orderer.genesis.block
    LocalMSPDir: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/msp

    LocalMSPID: OrdererMSP

    Profile:
        Enabled: false
        Address: 0.0.0.0:6060

    BCCSP:

        Default: SW
        SW:
            Hash: SHA2
            Security: 256
            FileKeyStore:
                KeyStore:

    FileLedger:
        Location: /opt/hyperledger/order/production/orderer
        Prefix: hyperledger-fabric-ordererledger

    RAMLedger:
        HistorySize: 1000

    Kafka:
        Retry:

```

```

ShortInterval: 5s
ShortTotal: 10m
LongInterval: 5m
LongTotal: 12h

NetworkTimeouts:
    DialTimeout: 10s
    ReadTimeout: 10s
    WriteTimeout: 10s

Metadata:
    RetryBackoff: 250ms
    RetryMax: 3

Producer:
    RetryBackoff: 100ms
    RetryMax: 3
Consumer:
    RetryBackoff: 2s
Verbose: false
TLS:
    Enabled: false
    PrivateKey:
    Certificate:
    RootCAs:

Version: 0.10.2.0
Debug:
    BroadcastTraceDir:
    DeliverTraceDir:

```

这两种方式都可以启动 orderer。在具体操作中，如果是通过 Docker 镜像文件的方式启动 orderer，推荐使用环境变量的配置方式；如果是采用命令直接启动的方式，推荐将所有信息都存放到配置文件中。

### 3. orderer 模块配置文件详解

orderer 模块的配置文件一共由 5 个部分组成，分别是：General、FileLedger、RAMLedger、Kafka、Debug。下面将分别介绍这 5 个部分的配置信息。

#### (1) General 节点相关的配置

General 节点中包含了 orderer 模块的基本控制信息。General 节点的配置示例如下：

```

General:
    LedgerType: file
    ListenAddress: 0.0.0.0
    ListenPort: 7050
    TLS:
        Enabled: false
        PrivateKey: server.key

```



```

Certificate: server.crt
RootCAs:
  - ca.crt
ClientAuthEnabled: false
ClientRootCAs:
LogLevel: debug
LogFormat:
GenesisMethod: file
GenesisProfile: TestOrgsOrdererGenesis
GenesisFile: orderer.genesis.block
LocalMSPDir: msp
LocalMSPID: OrdererMSP
Profile:
  Enabled: false
  Address: 0.0.0.0:6060

BCCSP:

Default: SW
SW:
  Hash: SHA2
  Security: 256
  FileKeyStore:
    KeyStore:

```

General 节点配置项的详细注释如下所示:

- **LedgerType**: 账本的类型, 有 ram、json、file 三种类型可以选择。ram 表示账本的数据保存在内存中, 一般用于测试环境。json 和 file 表示账本数据保存在文件中, 在生产环境中一般推荐使用 file。
- **ListenAddress**: orderer 服务器监听的地址, 如果服务器有多个网卡, 一般需要指明监听的具体地址。
- **ListenPort**: 监听端口。
- **Enabled**: 启用 TLS 时的相关配置。
- **PrivateKey**: 私钥文件。
- **Certificate**: 证书文件。
- **RootCAs**: 根证书文件。
- **ClientAuthEnabled**: 启用客户端证书验证。
- **ClientRootCAs**: 客户端根证书。
- **LogLevel**: 日志级别。
- **LogFormat**: 日志格式。
- **GenesisMethod**: 初始块的来源方式, 支持 provisional 或 file, provisional 表示 Genesis Profile 选项指定的内容在默认的配置文件中的配置是自动生成的, 后者使用 GenesisFile

指定的现成初始的文件。

- GenesisProfile: 初始块的 profile, 在 configtxgen 模块的配置文件中指定。
- GenesisFile: 初始块文件的路径。
- LocalMSPDir: orderer 模块 msp 文件的路径。
- LocalMSPID: orderer 模块的编号, 在 configtxgen 模块的配置文件中指定。
- Enabled: 是否启动 go 的 profile 信息。
- Address: go 的 profile 信息的访问地址。
- Default: 采用的密码机制, SW 为软件程序实现, PKCS11 为硬件的实现方式。
- Hash: 算法类型。

## (2) FileLedger 节点相关的配置

FileLedger 节点中包含了 orderer 模块中账本文件相关的配置信息。FileLedger 节点的配置示例如下:

```
FileLedger:
  Location: /opt/hyperledger/order/production/orderer
  Prefix: hyperledger-fabric-ordererledger
```

FileLedger 节点配置项的详细注释如下所示:

- Location: 账本文件的路径。
- Prefix: 账本存放在临时目录时候的目录名, 如果已经指定了 Location 的值, 则该选项无效。

## (3) RAMLedger 节点相关的配置

RAMLedger 节点中包含了 orderer 模块的账本在内存中数据保存方式的相关配置信息。RAMLedger 节点的配置示例如下:

```
RAMLedger:
  HistorySize: 1000
```

RAMLedger 节点配置项的详细注释如下所示:

- HistorySize: 如果 LedgerType 类型为 RAM 时内存中保存的区块的数目, 超过这个数目的区块将被放弃。

## (4) Kafka 节点相关的配置

Kafka 节点中包含了 orderer 模块中连接 Kafka 相关的信息, 注意: 如果 orderer 节点的排序模式选择了 solo, 那么该节点的所有配置均无效。Kafka 节点的配置示例如下:

```
Kafka:
  Retry:
    ShortInterval: 5s
    ShortTotal: 10m
```

```

LongInterval: 5m
LongTotal: 12h
NetworkTimeouts:
  DialTimeout: 10s
  ReadTimeout: 10s
  WriteTimeout: 10s
Metadata:
  RetryBackoff: 250ms
  RetryMax: 3
Producer:
  RetryBackoff: 100ms
  RetryMax: 3
Consumer:
  RetryBackoff: 2s
Verbose: false
TLS:
  Enabled: false
  PrivateKey:
  Certificate:
  RootCAs:
Version: 0.10.2.0

```

Kafka 节点配置项的详细注释如下所示:

- **Retry:** 如果 orderer 在启动的时候, Kafka 还没有启动或者 Kafka 宕机时重试的次数。
  - **ShortInterval:** 操作失败短重试状态下重试的时间间隔。
  - **ShortTotal:** 短重试状态下最多重试的时间。
  - **LongInterval:** 长重试状态下重试的时间间隔。
  - **LongTotal:** 长重试状态下最多重试时间。
  - **DialTimeout:** 等待超时时间。
  - **ReadTimeout:** 读超时时间。
  - **WriteTimeout:** 写超时时间。
  - **RetryBackoff:** Kafka 集群选举 leader 的元数据。
  - **RetryMax:** Kafka 集群选举 leader 的元数据。
  - **RetryBackoff:** 者消息超时时间。
  - **Verbose :** Kafka 客户端的日志级别, 在 orderer 的运行日志中显示 Kafka 的日志信息。
  - **Enabled:** 是否启动 TLS。
  - **PrivateKey:** 私钥前面。
- certificate: Kafka 的证书。
- rootCAs: 验证 Kafka 的根证书。
- version: Kafka 的版本号 (本书完稿时, Fabric 最新源代码中的默认版本是 0.10.2.0)。



### (5) Debug 节点相关的配置

Debug 节点中包含了 orderer 模块调试相关的选项。Debug 节点的配置示例如下：

```
Debug:
    BroadcastTraceDir:
    DeliverTraceDir:
```

## 5.2.5 peer

peer 模块是 Fabric 中最重要的模块，也是在 Fabric 系统使用最多的模块。peer 模块在 Fabric 中被称为主节点模块，主要负责存储区块链数据、运行维护链码、提供对外付服务接口等作用。本节将详细介绍 peer 模块的命令行参数、配置方式、启动方式等功能。

### 1. 命令行和常用参数

peer 模块中常用的命令和参数如下所示：

```
// 命令
chaincode    chaincode 相关操作  相关子命令  install|instantiate|invoke|package|query
              |signpackage|upgrade|list.
channel      channel 相关操作： create|fetch|join|list|update.
logging      Log levels: getlevel|setlevel|revertlevels.
node         启动 peer 节点服务器
version      显示当前 peerf 服务器的版本
```

参数：

```
--logging-level string    日志级别
--test.coverprofile string 测试配置文件
-v, --version             显示当前服务器的版本
```

### 2. peer 的环境变量

配置文件和环境变量是设置 peer 启动参数的重要手段，它们之间的关系在本章开始部分已经介绍了。peer 模块通过环境配置属性信息的示例如下所示：

```
export set CORE_VM_ENDPOINT=unix:///var/run/docker.sock
export set CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=host
export set CORE_PEER_PROFILE_ENABLED=true
export set CORE_LOGGING_LEVEL=debug
export set CORE_PEER_ID=peer0.org1.qklszzn.com
export set CORE_PEER_GOSSIP_USELEADERELECTION=true
export set CORE_PEER_GOSSIP_ORGLEADER=false
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_CHAINCODELISTENADDRESS=peer0.org1.qklszzn.com:7052
export set CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.qklszzn.com:7051
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/msp
export set CORE_PEER_TLS_ENABLED=false
export set CORE_PEER_TLS_CERT_FILE=/opt/hyperledger/fabricconfig/crypto-
```

```
config/peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/server.
crt
export set CORE_PEER_TLS_KEY_FILE=/opt/hyperledger/fabricconfig/crypto-config/
peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/server.key
export set CORE_PEER_TLS_ROOTCERT_FILE=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/ca.crt
```

### 3. peer 模块的配置文件

peer 的配置文件的默认文件名为 `croe.yaml`, 配置文件分为 `logging`、`peer`、`vm`、`chaincode`、`ledger` 这五大部分。各部分的详细含义如下所示。

#### (1) 配置文件中 `logging` 相关的属性

`logging` 节点定义了 `peer` 模块中所有模块的日志级别和日志格式, 每个模块的日志级别可以根据业务的需求定义成不一样的格式:

```
logging:
  peer:      debug
  cauthdsl:  warning
  gossip:    warning
  ledger:    info
  msp:       warning
  policies:  warning
  grpc:      error
  format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
```

日志分为 `critical`、`error`、`warning`、`notice`、`info`、`debug` 这六个级别。

#### (2) `peer` 节点相关的配置

`peer` 节点定义了 `peer` 模块一般的配置信息:

```
peer:
  id: peer0.org1.qklszzn.com
  networkId: dev
  listenAddress: 0.0.0.0:7051
  chaincodeListenAddress: 0.0.0.0:7052
  address: peer0.org1.qklszzn.com:7051
  addressAutoDetect: false
  gomaxprocs: -1

  gossip:
    bootstrap: 127.0.0.1:7051
    useLeaderElection: true
    orgLeader: false
    endpoint:
    maxBlockCountToStore: 100
```

```

maxPropagationBurstLatency: 10ms
maxPropagationBurstSize: 10
propagateIterations: 1
propagatePeerNum: 3
pullInterval: 4s
pullPeerNum: 3
requestStateInfoInterval: 4s
publishStateInfoInterval: 4s
stateInfoRetentionInterval:
publishCertPeriod: 10s
skipBlockVerification: false
dialTimeout: 3s
connTimeout: 2s
recvBuffSize: 20
sendBuffSize: 200
digestWaitTime: 1s
requestWaitTime: 1s
responseWaitTime: 2s
aliveTimeInterval: 5s
aliveExpirationTimeout: 25s
reconnectInterval: 25s
externalEndpoint: peer0.org1.qklszzn.com:7051

```

```

election:
  startupGracePeriod: 15s
  membershipSampleInterval: 1s
  leaderAliveThreshold: 10s
  leaderElectionDuration: 5s

```

```

pvtData:
  maxPeers: 3
  minAck: 3

```

events:

```

address: 0.0.0.0:7053
bufferSize: 100
timeout: 10ms

```

tls:

```

enabled: false
cert:
  file: server.crt
key:
  file: server.key
rootcert:
  file: ca.crt

```

serverhostoverride:



## 88 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```
fileSystemPath: /opt/hyperledger/peer/production

BCCSP:
  Default: SW
  SW:
    Hash: SHA2
    Security: 256
    FileKeyStore:
      KeyStore:

mspConfigPath: msp
localMspId: Org1MSP

profile:
  enabled: false
  listenAddress: 0.0.0.0:6060

handlers:
  authFilter: "DefaultAuth"
  decorator: "DefaultDecorator"
```

peer 节点的选项比较多, 下面将分类介绍。

第一类: 通用属性

- id: peer 节点的编号;
- networkId: peer 节点的网络编号;
- listenAddress: peer 节点的监听地址;
- chaincodeListenAddress: chaincode 的监听地址;
- address: 访问地址;
- addressAutoDetect: 锚节点地址;
- gomaxprocs: 最大有效数;
- filePath: 区块等数据的存放路径;
- mspConfigPath: 当前节点 MSP 文件的路径。

第二类: gossip

- bootstrap: 启动节点后向哪些节点发起 gossip 连接, 以加入网络。这些节点与本地节点需要属于同一组织;
- endpoint: 本节点在同一组织内的 gossip id, 默认为 peer.address;
- useLeaderElection: 用户组织节点的生成方式;
- orgLeader: 当前节点是否为用户组织节点;
- maxBlockCoimtToStore: 保存到内存中的区块个数上限, 超过则丢弃;

- `maxPropagationBurstLatency`: 保存消息的最大时间, 超过则触发转发给其他节点;
- `maxPropagationBurstSize`: 保存的最大消息个数, 超过则触发转发给其他节点;
- `propagateIterations`: 消息转发的次数;
- `propagatePeerNum`: 推送消息给指定个数的节点;
- `pullInterval`: 拉取消息的时间间隔;
- `pullPeerNum`: 从指定个数的节点拉取消息;
- `requestStateInfoInterval`: 从节点拉取状态信息 (StateInfo) 消息的间隔;
- `publishStateInfoInterval`: 向其他节点推送状态信息消息的间隔;
- `publishCertPeriod`: 启动后, 在心跳消息中嵌入证书的等待时间;
- `stateInfoRetentionInterval`: 状态信息消息的超时时间;
- `skipBlockVerification`: 是否不对区块消息进行校验, 默认为 `false`;
- `dialTimeout`: gRPC 连接拨号的超时时间;
- `connTimeout`: 建立连接的超时时间;
- `recvBuffSize`: 收取消息的缓冲大小;
- `sendBuffSize`: 发送消息的缓冲大小;
- `digestWaitTime`: 处理摘要数据的等待时间;
- `requestWaitTime`: 处理 `nonce` 数据的等待时间;
- `responseWaitTime`: 终止拉取数据处理的等待时间。
- `aliveTimeInterval`: 定期发送 `Alive` 心跳消息的时间间隔;
- `aliveExpirationTimeout`: `Alive` 心跳消息的超时时间;
- `reconnectInterval`: 断线后重连的时间间隔;
- `externalEndpoint`: 节点被组织外节点感知时的地址, 默认为空, 代表不被其他组织所感知。

### 第三类: events

- `address`: 事件监听器地址;
- `bufferSize`: 事件消息缓存数, 超过该值会被阻塞;
- `timeout`: 队列阻塞的超时时间。

### 第四类: tls

- `enabled`: 是否激活 `tls`;
- `cert`: 服务身份验证证书;
- `key`: 服务的私钥文件;
- `rootcert`: 根服务器证书;
- `serverhostoverride`: `tls` 握手时候制定服务名称。

### 第五类: BCCSP

BCCSP 主要配置加密和解密类的相关信息, 具体可以参考相关文档。

### (3) vm 节点相关的配置

vm 节点定义 peer 和 docker 交互的相关配置如下:

```
vm:
  endpoint: unix:///var/run/docker.sock
  docker:
    tls:
      enabled: false
      ca:
        file: docker/ca.crt
      cert:
        file: docker/tls.crt
      key:
        file: docker/tls.key
    attachStdout: false
  hostConfig:
    NetworkMode: host
    Dns:

    LogConfig:
      Type: json-file
      Config:
        max-size: "50m"
        max-file: "5"
    Memory: 2147483648
```

vm 节点配置项的详细注释如下所示:

- endpoint: docker 服务器 Daemon 的地址, 默认取端口的套接字;
- tls: 启动 docker 的 tls 证书;
- attachStdout: 是否将 docker 消息绑定到指定的输出;
- NetworkMode: chaincode 容器的网络命名模式;
- Dns: 是否启用域名服务器;
- LogConfig: docker 容器的日志配置信息;
- Type: 日志类型;
- Memory: 占用内存。

### 4. 配置文件中 chaincode 节点相关的配置

chaincode 定义了链码相关的配置, 如下所示:

```
chaincode:
  peerAddress:
  id:
```

```

path:
name:

builder: $(DOCKER_NS)/fabric-ccenv:$(ARCH)-$(PROJECT_VERSION)
golang:
    runtime: $(BASE_DOCKER_NS)/fabric-baseos:$(ARCH)-$(BASE_VERSION)
car:
    runtime: $(BASE_DOCKER_NS)/fabric-baseos:$(ARCH)-$(BASE_VERSION)

java:
    Dockerfile: |
        from $(DOCKER_NS)/fabric-javaenv:$(ARCH)-$(PROJECT_VERSION)
node:
    runtime: $(BASE_DOCKER_NS)/fabric-baseimage:$(ARCH)-$(BASE_VERSION)

startuptimeout: 300s
executetimeout: 30s
mode: net
keepalive: 0
system:
    csc: enable
    lsc: enable
    esc: enable
    vsc: enable
    qsc: enable
    rsc: disable
logging:
    level: info
    shim: warning
    format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'

```

chaincode 节点配置项的详细注释如下所示:

- peerAddress: chaincode 中的 peer 服务器地址;
- builder: 本地的编译环境为 docker 镜像;
- golang: Go 语言版的 chaincode 的基础镜像;
- car: car 格式的 chaincode 生成镜文件时的基础镜像;
- java: Java 语言版的 chaincode 的基础镜像;
- node: Node 语言版的 chaincode 的基础镜像;
- startuptimeout: 启动 chaincode 容器时的超时时间, 超过这个时间认为启动失败;
- executetimeout: 执行 Invoke 和 Init 方法时的超时时间, 超过这个时间认为执行失败;
- mode: chaincode 的运行模式, net 为网络模式, dev 为开发模式, dev 模式下, 可以在容器外运行 chaincode;
- keepalive: peer 节点和 chaincode 直接的心跳时间;



- system: 系统 chaincode 的开关;
- logging: chaincode 的日志级别。

## 5. 配置文件中 ledger 节点相关的配置

ledger 节点定义了账本相关的配置, 如下所示:

```
ledger:
  blockchain:
    state:
      stateDatabase: goleveldb
      couchDBConfig:
        couchDBAddress: 127.0.0.1:5984
        username:
        password:
        maxRetries: 3
        maxRetriesOnStartup: 10
        requestTimeout: 35s
        queryLimit: 10000
  history:
    enableHistoryDatabase: true
```

ledger 节点配置项的详细注释如下所示:

- state: 状态存储数据库的配置;
- stateDatabase: 数据库类型, 目前支持 goleveldb 和 CouchDB。
- couchDBConfig: stateDatabase 的类型为 CouchDB, CouchDB 相关的参数;
- enableHistoryDatabase: 是否保存状态的历史数据库, 生产系统中建议开启。

## 5.3 Fabric 模块在系统中的作用

前面我们知道 Fabric 是由多个模块组成的程序组, 其中 orderer 和 peer 这两个模块是 Fabric 的核心模块。为了说明 orderer 模块和 peer 模块在 Fabric 系统中的作用, 下面我们通过图 5-1 所示的 Fabric 系统体系结构图来说明这两个模块在整个 Fabric 系统占有的核心地位。

### 5.3.1 peer 模块在 Fabric 系统中的作用

图 5-1 中列出了 peer 模块和 orderer 模块在 Fabric 系统中的作用, 从图中我们会发现, 在一个组织内会出现 4 个 Peer 服务器节点, 这 4 个 Peer 服务器节点并不是 4 个 Peer 程序进程, 而是表示一个组织中的 4 个角色。这一个逻辑概念并不代表每个具体的 Peer 程序进程, 但是一个完整的组织必须具备这 4 个角色, 即使组织内只有一个 Peer 程序进程, 依然具备这样 4 个角色, 只是这 4 个角色由一个 Peer 服务器节点兼任。当然有些节点也可以存在多

个，比如 Commit 节点和 Endorse 节点是可以存在多个的。但是 Leader 节点和 Anchor 节点只能存在一个。下面将分别介绍这四个角色。

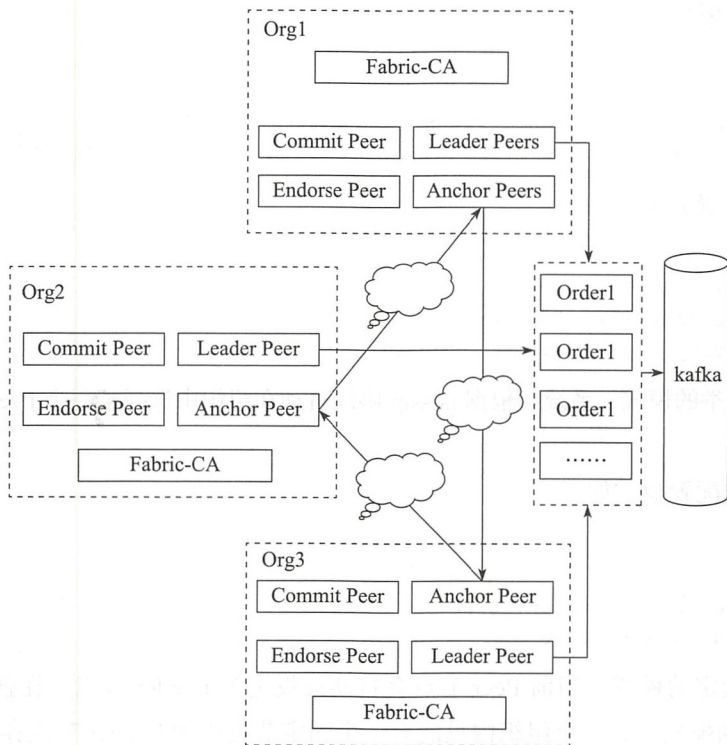


图 5-1 Fabric 系统中 peer 模块和 orderer 模块的架构图

### 1. 提交节点 (Committer Peer)

Committer 节点主要负责维护区块链的账本结构，该节点会定期从 Orderer 节点获取包含交易的区块，在对这些区块链进行合法发行校验之后，会把这些区块链加入到区块链中。Committer 节点无法通过配置文件配置，当前客户端或者命令行发起交易请求的时候需要指定相关的 Committer 节点。

### 2. 背书节点 (Endorse Peer)

Endorse 节点主要负责对交易进行校验。当 Endorse 节点接收到客户端发送的交易请求之后会对交易的合法性进行校验，校验成功之后会将结果反馈给客户端。Endorse 节点也是无法通过配置文件指定的，而是由发起交易请求的客户端指定的。Endorse 节点在组织中可以有多个。

### 3. Leader 节点 (Leader Peer)

Leader 节点是负责代表组织从 Orderer 节点中获取区块信息。Leader 节点在整个组织中

只有一个。Endorse 节点的产生方式是通过 peer 模块的配置文件 core.yaml 进行配置的。在配置文件有两种方式制定 Leader 节点, 分别是: 自主选举和强制制定。Leader 节点相关的配置信息如下所示:

```
peer:
  gossip:
    useLeaderElection: true // 是否自动选举 leader 节点
    orgLeader: false      // 是否代表组织从 Orderer 获取区块链数据
```

自主选举配置如下:

```
peer:
  gossip:
    useLeaderElection: true
    orgLeader: false
```

选择自动选举的模式, 系统会根据 gossip 协议自动在组织中选择某一个 Peer 节点为 Leader 节点。

强制制定的配置模式如下:

```
peer:
  gossip:
    useLeaderElection: true
    orgLeader: false
```

选择强制制定的模式, 当前 Peer 节点会自动被设定为 Leader 节点。在强制指定方式设定 Leader 节点的模式中, 一个组织内只能有一个固定节点作为 Leader 节点存在。

在实际的项目中, 我们建议采用自主选举的模式。因为在该模式下, 如果被选举的 Leader 节点发生故障, 系统会自动选取另外一个节点作为 Leader 节点。

#### 4. 锚节点 (Anchor Peer)

锚节点主要负责代表组织和其他组织进行信息交换。每个组织都有一个锚节点, 锚对于组织来说非常重要, 如果锚节点出现问题, 那么组织和其他组织会失去联系。锚节点也是通过配置的方式指定。锚节点的配置信息在 configtxgen 模块的配置文件 configtx.yaml 中配置的。configtx.yaml 文件中关于锚节点的配置信息如下:

```
Organizations:
  - &Org1
  AnchorPeers:
    - Host: peer0.org1.qklszzn.com
      Port: 7051
```

- Host 属性: Host 属性表示本组织锚节点的访问地址, 可以是 IP 地址或者域名, 建议采用域名。

- Port 属性: Port 属性表示本组织锚节点的访问端口。

任何组织都必须保证本组织内锚节点服务器处于可访问的状态。

### 5.3.2 orderer 模块在 Fabric 系统中的作用

在 Fabric 中, orderer 模块负责对不同客户就端发送的交易进行排序和打包。目前 Orderer 组件提供了两种打包模式: Solo 模式和 Kafka 模式。其中 Solo 模式太简单仅适用于开发模式,在生产系统中推荐使用 kafka 模式。orderer 模块的工作原理如下:

- 客户端向 orderer 模块发送交易。
- Orderer 节点对交易进行检查,如果符合条件,则将交易发送到排序队列(Solo 模式在本地,Kafka 模式会提交给 Kafka)。
- Orderer 节点从消息队列中对取出交易并进行打包。打包之后会将相关的消息存储到本地存储中。
- Orderer 节点根据客户端代码请求,将区块链发送给客户端。

## 5.4 Fabric 数据安全传输的方式

为了保证数据传输中的安全性, Fabric 提供了一系列相关的组件和配置,这些组件和配置在 Fabric 中被统称为 Transport Layer Security (TLS)。TLS 不是 Fabric 中的必选项,可以通过相关的配置参数激活或者关闭。因为 TLS 在 Fabric 的部署和维护的过程中非常容易引发误解和疑惑,为此本书专门设置一个小节来讨论 Fabric 中 TLS 相关的内容。

通过第 4 章我们知道 Fabric 一共包含 5 个核心模块,在这 5 个核心模块中 peer 模块和 orderer 模块具有 TLS 和相关的设置。

### 5.4.1 Fabric 中 orderer 模块 TLS 设置

orderer 模块中 TLS 相关的设置主要包括配置文件和环境变量两种方式。

1) orderer 模块在配置文件中设置 TLS 的样例如下所示:

```
General:
  TLS:
    Enabled: true
    PrivateKey: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/server.key
    Certificate: /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/server.crt
    RootCAs:
      - /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/tls/ca.crt
```



```
ClientAuthEnabled: false
ClientRootCAs:
```

2) orderer 模块在环境变量中设置 TLS 的样例如下所示:

```
ORDERER_GENERAL_TLS_ENABLED = true
ORDERER_GENERAL_TLS_PRIVATEKEY = /opt/hyperledger/fabricconfig/crypto-config/
ordererOrganizations/qklszzn.com/orderers/orderer.qklszzn.com/tls/server.key
ORDERER_GENERAL_TLS_CERTIFICATE = /opt/hyperledger/fabricconfig/crypto-config/
ordererOrganizations/qklszzn.com/orderers/orderer.qklszzn.com/tls/server.crt
```

3) orderer 模块 TLS 相关配置的详细解释如下:

- General.TLS.Enabled: TLS 激活标记, true 表示激活, false 表示关闭。
- General.TLS.PrivateKey: 服务器私钥文件路径。
- General.TLS.Certificate: 服务器数字证书文件路径。
- General.TLS.RootCAs: 根 CA 服务器证书文件的路径。

4) orderer 模块 TLS 配置中相关文件的路径如下:

在上述 orderer 模块 TLS 的配置中涉及的相关证书文件包含在 cryptogen 模块生成的 orderer 模块的账号文件夹中, 可以参见本书第 4 章生成的 orderer 模块的证书文件。进入 orderer 模块相关证书文件所在的文件夹, 执行命令 `tree -L 4` 显示内容如下:

```
├── orderer.robertfabrictest.com
│   ├── msp
│   │   ├── admincerts
│   │   ├── cacerts
│   │   ├── keystore
│   │   ├── signcerts
│   │   └── tlscacerts
│   └── tls
│       ├── ca.crt
│       ├── server.crt
│       └── server.key
```

在文件夹 `tls` 中存放 orderer 模块 TLS 设置需要用到的相关证书文件。

## 5.4.2 Fabric 中 peer 模块 TLS 设置

peer 模块中 TLS 相关的设置主要包括配置文件和环境变量两种方式。但是由于 peer 模块有服务器节点 (peer node 命令) 和命令行接口两种运行模式, 这两种模式的运行方式有些区别, 下面我们将分别介绍不同模式下 peer 模块 TLS 相关的设置方式。

### 1. peer 模块作为服务器节点运行时 TLS 的设置

peer 模块作为服务器节点运行时, 可以通过配置和环境变量这两种方式设置 TLS 相关的配置信息, 这两种方式分别如下所示:

### (1) 配置文件中 TLS 属性的设置方式

```
peer:
  tls:
    enabled: true
    cert:
      file: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/server.crt
    key:
      file: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/server.key
    rootcert:
      file: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/ca.crt
    serverhostoverride:
```

### (2) 环境变量中 TLS 属性的设置方式

```
CORE_PEER_TLS_ENABLED=false
CORE_PEER_TLS_CERT_FILE=/opt/hyperledger/fabricconfig/crypto-config/
peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/server.crt
CORE_PEER_TLS_KEY_FILE=/opt/hyperledger/fabricconfig/crypto-config/
peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/server.key
CORE_PEER_TLS_ROOTCERT_FILE=/opt/hyperledger/fabricconfig/crypto-config/
peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/ca.crt
```

相关配置选项的详细解释如下所示：

- peer.tls.enabled: TLS 激活标记, true 表示激活, false 表示关闭。
- peer.tls.privateKey: 服务器私钥文件路径。
- peer.tls.certificate: 服务器数字证书文件路径。
- peer.tls.rootCAs: 根 CA 服务器证书文件的路径。

在上述 peer 模块 TLS 相关配置中涉及的相关证书文件包含在 cryptogen 模块生成的 Peer 模块的账号文件夹中, 可以参见本书第 4 章生成的 peer 模块的证书文件。进入 peer 模块相关证书文件所在的文件夹, 执行命令 `tree -L 4` 显示内容如下:

```
├── peer0.org1.qklszzn.com
│   ├── msp
│   │   ├── admincerts
│   │   ├── cacerts
│   │   ├── keystore
│   │   ├── signcerts
│   │   └── tlscacerts
│   └── tls
│       ├── ca.crt
│       ├── server.crt
│       └── server.key
```

在文件夹 `tls` 中存放 peer 模块 TLS 设置需要用到相关证书文件。

## 2. peer 模块采用命令行工具的方式运行时 TLS 属性的设置

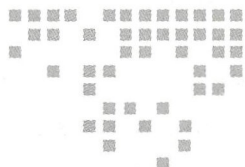
peer 模块的命令行选项中有一些操作需要和远程的 peer 模块服务器节点或者 orderer 模块进行通信,这个时候可以选择采用 TLS 的方式以提高安全性。以 peer 模块的 Chaincode 选项为例,采用 TLS 方式实例化 Chaincode 的命令格式如下所示:

```
peer chaincode instantiate -o om:7050 -C roberttestchannel -n r_test_cc6 -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')" --tls --cafile /opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/qklszzn.com/orderers/orderer.qklszzn.com/msp/tlscacerts/tlsca.qklszzn.com-cert.pem
```

通过上述命令我们可以发现 peer 模块命令行的运行方式中可以通过 --tls 和 --cafile 这两个命令行参数来确定,其中 --cafile 参数所使用的文件和 orderer 模块是相同的。

## 5.5 本章小结

本章简单介绍了 Fabric 的 5 个核心模块的使用方法和相关的配置信息。通过本章,读者可以详细了解 Fabric 各模块的技术特性和使用方法,为后续内容的阅读打下基础。在本书的第 10 章中我们会以一个项目案例的方式介绍 Fabric 的核心模块在 Fabric 项目流程中如何使用,读者可以根据自己对 Fabric 的了解选择阅读。



## Fabric 的账号体系

Fabric 的账号体系是 Fabric 的重要组成部分，由于 Fabric 是基于证书而不是传统的用户名和密码形式的身份和角色认证的，因此很多从事传统基于数据库系统开发的技术人员在转向 Fabric 开发时会遇到很多困惑。本章内容将详细介绍 Fabric 的账号体系，帮助相关开发人员解决与 Fabric 账号证书体系相关的困惑。

### 6.1 Fabric 账号简介

在任何非开放系统中都需要通过账号和密码对系统入口进行相关的管理。比如我们常用的电子邮件系统都需提供账号和密码方可使用，再比如常见的数据库管理系统也需要获取相应的账号和密码才能对数据库进行操作。通过本章前面的内容我们知道 Fabric 是一个联盟链，联盟链的特点是用户非授权时不能接入区块链。因此 Fabric 系统中存在一套授权体系，我们将这个体系称为 Fabric 的账号体系。关于 Fabric 的账号我们要搞清楚两个问题：

- Fabric 账号是什么。
- 什么样的操作需要使用到 Fabric 的账号。

#### 6.1.1 Fabric 账号是什么

Fabric 中的账号实际上是根据 PKI 规范生成的一组证书和秘钥文件。在 5.2.1 节中，cryptogen 模块生成的文件中就包含 Fabric 账号相关的证书文件。我们通常接触到的账号系统一般是由账号和密码两个属性组成的，比如常用的电子邮箱系统。在这些系统中账号和密

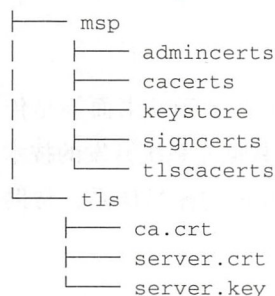


码只是获取操作权限的工具,一旦账号和密码验证成功,后面的操作基本上就和账号密码没有什么关系了。

但是区块链系统的一个非常重要的特点是:记录在区块链中的数据具有不可逆、不可篡改的特性。在 Fabric 中每条交易都会加上发起者的标签(签名证书),同时用发起人的私钥进行加密。如果交易需要其他组织的节点提供背书功能,那么背书节点也会在交易中加入自己的签名。这样每一笔交易的操作过程会非常清晰并且不可篡改。鉴于传统系统中基于账号和密码的验证体系已经无法完成这样的工作,因此 Fabric 设计了基于 PKI 规范的账号系统满足这样的要求。

既然 Fabric 的账号如此重要并且功能强大,同时又和传统的证书体系有本质的区别,那么它到底是什么样子?又由那些部分组成呢?我们通过下面的例子详细了解 Fabric 账号中包含哪些部分。以 5.2.1 节中 cryptogen 模块生成账号为例,截取其中一个完整的账号包含的内容来说明 Fabric 账号的结构。

一个完整的 Fabric 账号中包含的内容如下所示:



上述示例中的所有文件公共构成 Fabric 系统中的账号,通过上面的例子我们发现每个 Fabric 账号包含若干证书文件和秘钥文件,由此可见 Fabric 的账号体系比传统系统中由账号和密码组成的认证体系要复杂很多。仔细观察 Fabric 账号中证书文件的路径,我们发现这些证书分别存放在 msp 文件夹和 tls 文件夹中。

### 1. msp 文件夹中内容

msp 中主要存放签名用的证书文件和加密用的私钥文件。msp 中包含以下 5 个部分:

- admincerts: 管理员证书。
- cacerts: 根 CA 服务器的证书。
- keystore: 节点或者账号的私钥。
- signcerts: 符合 X.509 的节点或者用户证书文件。
- tlscacerts: TLS 根 CA 的证书。

### 2. tls 文件夹中的内容

tls 文件夹中存放加密通信相关的证书文件。这一组文件实际上起到了账号的作用。我

们把这样一组文件称为 Fabric 的账号。

账号的说法是本书对 Fabric 中账号的特定称谓。Fabric 的官方文档把这些证书称为 Membership Service Providers, 简称为 MSP, 我们无法找到更加准确的翻译, 所以本书暂且用账号这个比较传统的名词来指代 Fabric 生成的这些证书。从实际的运行效果看, 这些证书文件等同于账号的作用。

这里最后再啰嗦一遍, 本书中账号就是 MSP, 本书中账号就是 MSP, 本书中账号就是 MSP。重要的事情需要重复三遍。

### 6.1.2 什么地方需要使用 Fabric 的账号

Fabric 中 Orderer、Peer、客户端 SDK、CLI 接口等所有操作都需要账号。Fabric 中每个具体的动作, 创建通道、部署 chaincode、调用 chaincode 等都需要指定的账号。每个 Peer 向 Orderer 发送请求的时候也需要 Peer 的账号。在 Fabric 中如果需要新增加一个 Peer 节点, 首先做的事情是给这个 Peer 创建账号。为了显示账号的重要性, 下面列举一些 Fabric 中常用的操作, 看看这些操作是如何使用账号的。

这里的目录是基于本书作者的演示环境的目录, 读者可以根据自己的环境查找相关的文件。

#### 1. 启动 Orderer 需要的账号

启动 Orderer 的时候我们需要通过环境变量或者配置文件给当前启动的 Orderer 设定相应的账号。

环境变量设置账号:

```
ORDERER_GENERAL_LOCALMSPDIR=/opt/hyperledger/fabricconfig/crypto-config/
ordererOrganizations/qklszzn.com/orderers/orderer.qklszzn.com/msp
```

配置文件为 General 节点下面的 LocalMSPDir 子节点, 如下所示:

```
General:
  LocalMSPDir:/opt/hyperledger/fabricconfig/crypto-config/ordererOrganizations/
qklszzn.com/orderers/orderer.qklszzn.com/msp
```

启动的 Orderer 的账号是什么样子, 我们进入相关目录执行命令 `tree -L 4`, 如下所示:

```
├── admincerts
│   └── Admin@qklszzn.com-cert.pem
├── cacerts
│   └── ca.qklszzn.com-cert.pem
└── keystore
```

```

|       └── fa8253c580a6d96bd542fb86b320cb92ad9d85ea336af21f2c0a93a3518c4d6c_sk
|── signcerts
|       └── orderer.qklszzn.com-cert.pem
|── tlscacerts
|       └── tlsca.qklszzn.com-cert.pem

```

具体的内容可以参考 6.1 节中的说明。

## 2. 启动 Peer 需要的账号

启动 Peer 的时候我们需要通过环境变量或者配置文件给当前启动的 Peer 设定相应的账号。

环境变量设置账号:

```
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/msp
```

配置文件为 peer 节点下面的 mspConfigPath 子节点, 如下所示:

```
peer:
  mspConfigPath:/opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/peers/peer0.org1.qklszzn.com/msp
```

启动的 Peer 的账号是什么样子, 我们进入相关目录执行命令 `tree -L 4`, 如下所示:

```

|── admincerts
|   └── Admin@org1.qklszzn.com-cert.pem
|── cacerts
|   └── ca.org1.qklszzn.com-cert.pem
|── keystore
|   └── 4cb0b17fdc61d192984ceb600c90adc4420b5ecec13fe1bc6bc3752ece187e6_sk
|── signcerts
|   └── peer0.org1.qklszzn.com-cert.pem
|── tlscacerts
|   └── tlsca.org1.qklszzn.com-cert.pem

```

具体的内容可以参考 6.1 节中是说明。

## 3. 创建 Channel 需要用到的账号

Channel 是 Fabric 中非常重要的组成部分, 创建 Channel 的时候也是需要账号的。读者可以参考 5.2.6 节创建账号的脚本, 其中有一个环境变量 `CORE_PEER_MSPCONFIGPATH`, 该环境变量的设置如下:

```
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

`CORE_PEER_MSPCONFIGPATH` 环境变量设置了创建当前账号的用户信息, 我们可以通过命令 `tree -L 4` 进入该目录看以下账号的内容:

```

├── admincerts
│   └── Admin@org1.qklszzn.com-cert.pem
├── cacerts
│   └── ca.org1.qklszzn.com-cert.pem
├── keystore
│   └── b031338f76290f089d330b064d4534202a49ae8d65ca5d266c377bc46812a884_sk
├── signcerts
│   └── Admin@org1.qklszzn.com-cert.pem
└── tlscacerts
    └── tlscacert.org1.qklszzn.com-cert.pem

```

具体的内容可以参考 6.1 节中的说明。

我们可以发现这些账号的内容是一样的，都包含了 5 个不同的文件，但是我们仔细观察会发现在文件路径上面还是有一些区别的。我们把路径中相同的部分隐藏后对比如下：

```

#Orderer 启动的账号路径
ordererOrganizations/qklszzn.com/orderers/orderer.qklszzn.com/msp

#Peer 启动的账号路径
peerOrganizations/org1.qklszzn.com/peers/peer0.org1.qklszzn.com/msp

# 创建 Channel 的账号路径
peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

```

我们可以发现 Peer 和 Orderer 都有属于自己的账号，创建 Channel 使用的是用户账号。其中 Peer 和创建 Channel 的用户账号属于某个组织，而 Orderer 的启动账号只属于他自己。这里特别注意，用户账号在很多操作中都会用到，而且很多操作的错误都是用户账号的路径设置不当而引起的。在本书后面需要设置用户账号的地方会详细说明用户账号的使用方法。

## 6.2 基于 cryptogen 的账号管理体系

cryptogen 模块是创建 Fabric 账号的方式之一，在 5.2.1 节中已经做了详细的介绍，这里不再重复，如果不清楚，可以参考 5.2.1 节的内容。本节中主要讨论 cryptogen 的局限性。

前面我们了解到 cryptogen 模块可以通过配置文件生成 Fabric 运行所需要的相关账号文件，在配置文件中可以指定每个组织包含的用户数和节点数，然后 cryptogen 模块会根据配置文件的定义生成相应数目的配置文件。但是细心的读者可能会发现一个问题，如果系统发生变化需要引入新的组织，或者组织中需要增加新的账号和用户，这个该如何处理呢？

cryptogen 模块开发者们显然已经考虑到这个问题，通过一个配合可以部分解决这些问题。首先通过 cryptogen 模块的子命令 showtemplate 显示默认的模板，默认模板内容如下所示：



```
OrdererOrgs:
```

```
- Name: Orderer
  Domain: example.com
  Specs:
    - Hostname: orderer
```

```
PeerOrgs:
```

```
- Name: Org1
  Domain: org1.example.com

  Template:
    Count: 1

  Users:
    Count: 1

- Name: Org2
  Domain: org2.example.com
  Template:
    Count: 1
  Users:
    Count: 1
```

在配置文件中 PeerOrgs 节点 Template 子节点中有个属性 Count, Count 表示当前组织包含 Peer 节点的数目, 同时也会生成相应数目的配置文件。在 6.2.1 节列举的例子中我们给组织 Org1 设置的 Peer 节点数为 4, 那么 cryptogen 模块根据配置文件会给组织 Org1 生成 4 个 Peer 节点所有需要的账号。进入组织 Org1 的账号文件夹中, 执行 tree -L 2 命令内容显示如下:

```
├── ca
│   ├── a1db721c0cfb6f107fc45501d29866633d21a3492c87bb352687b2e8e85b652e_sk
│   └── ca.org1.qklszzn.com-cert.pem
├── msp
│   ├── admincerts
│   ├── cacerts
│   └── tlscacerts
├── peers
│   ├── peer0.org1.qklszzn.com
│   ├── peer1.org1.qklszzn.com
│   ├── peer2.org1.qklszzn.com
│   └── peer3.org1.qklszzn.com
├── tlsca
│   ├── f92d78e8195795e3a115c4c129d78edaac70ec4f5ef4764df43e1de245e7d467_sk
│   └── tlsca.org1.qklszzn.com-cert.pem
└── users
    ├── Admin@org1.qklszzn.com
```

```

|—— User1@org1.qklszzn.com
|—— User2@org1.qklszzn.com
|—— User3@org1.qklszzn.com
|—— User4@org1.qklszzn.com
|—— User5@org1.qklszzn.com
|—— User6@org1.qklszzn.com

```

通过上面内容可以发现，一共生成了 4 个 Peer 节点账号文件，这些文件夹的命名是有规则的，peer+ 索引为前缀 + 域名。所以前缀的值为 0 到 Count 值 -1。这里有一个 start 属性没有赋值，start 属性就是继续添加 Peer 节点使用的。Start 用来表示生成 Peer 节点账号文件的文件夹命名的前缀，默认是 0。如果需要增加新的节点，可以给 Start 属性赋值上一次的 count。这里以 5.2.1 节的例子为例，如果需要给组织 Org1 新增加 3 个 Peer 节点，可以使用下面的配置：

```

PeerOrgs:
  - Name: Org1
    Domain: org1.qklszzn.com
    Template:
      Count: 3
      Start: 4

```

通过上面的配置文件，cryptogen 模块可以给组织 Org1 新增加三个 Peer 节点的配置文件，我们重新回到组织 Org1 的账号文件目录中，再次执行命令 `tree -L 2`，显示如下：

```

|—— ca
|   |—— aldb721c0cfb6f107fc45501d29866633d21a3492c87bb352687b2e8e85b652e_sk
|   |—— ca.org1.qklszzn.com-cert.pem
|—— msp
|   |—— admincerts
|   |—— cacerts
|   |—— tlscacerts
|—— peers
|   |—— peer0.org1.qklszzn.com
|   |—— peer1.org1.qklszzn.com
|   |—— peer2.org1.qklszzn.com
|   |—— peer3.org1.qklszzn.com
|   |—— peer4.org1.qklszzn.com
|   |—— peer5.org1.qklszzn.com
|   |—— peer6.org1.qklszzn.com
|—— tlsca
|   |—— f92d78e8195795e3a115c4c129d78edaac70ec4f5ef4764df43e1de245e7d467_sk
|   |—— tlsca.org1.qklszzn.com-cert.pem
|—— users
|   |—— Admin@org1.qklszzn.com
|   |—— User1@org1.qklszzn.com
|   |—— User2@org1.qklszzn.com
|   |—— User3@org1.qklszzn.com

```

```

|—— User4@org1.qklszzn.com
|—— User5@org1.qklszzn.com
|—— User6@org1.qklszzn.com

```

这里成功地添加了三个 Peer 节点的配置文件,但是细心的读者可能会发现还有一个问题没有解决,那就是如果需要动态地增加用户账号该怎么办呢?目前最新的 Fabric 版本中并没有提供相关的功能。其实通过上面的例子读者可以发现,即使是增加 Peer 节点的过程也是比较麻烦的,那有没有一个办法能让增加节点和用户账号的方法简单一点呢?当然是有的,下一节中我们将介绍 hyperledger 项目组中专门为解决 Fabric 账号问题而发起的项目 Fabric-ca。

### 6.3 Fabric 账号服务器: Fabric-ca

Fabric-ca 项目是专门为了解决 Fabric 账号问题而发起的一个开源项目,它非常完美地解决了 Fabric 账号生成的问题。Fabric-ca 项目由 Fabric-ca-server 和 Fabric-ca-client 这两个模块组成。其中 Fabric-ca-server 模块在 Fabric 项目占有非常重要的作用,具体可以参考图 6-1。

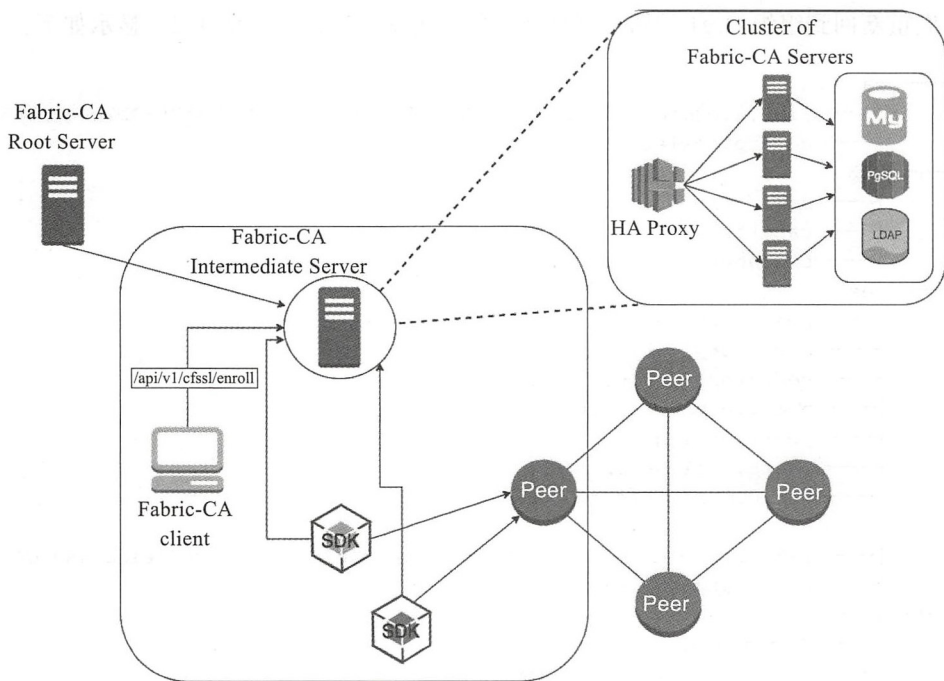


图 6-1 Fabric-ca 在 Fabric 系统中的作用示意图

### 6.3.1 Fabric-ca 的编译和安装

Fabric-ca 也是用 Golang 开发的应用程序，因此 Fabric-ca 的编译过程和 Fabric 其他组件的编译过程是类似的。下面将演示 Fabric-ca 在 Ubuntu 和 MacOS 系统上面的安装。

#### 1. Fabric-ca 在 Ubuntu 和 CentOS 上面的安装

第一步：安装系统组件包。

```
sudo apt install libtool libltdl-dev
```

如果在 CentOS 上面安装，则需要执行命令 `yum install libtool libltdl-dev`，其他的步骤是一样的。

第二步：下载源代码并编译。

```
cd $GOPATH/src/github.com/hyperledger

git clone http://gerrit.hyperledger.org/r/fabric-ca

cd $GOPATH/src/github.com/hyperledger/fabric-ca

make fabric-ca-server
make fabric-ca-client
```

第三步：安装编译好的可执行文件。

```
cd $GOPATH/src/github.com/hyperledger/fabric-ca/bin

cp $GOPATH/src/github.com/hyperledger/fabric-ca/bin/* /usr/local/bin

chmod -R 775 /usr/local/bin/fabric-ca-server
chmod -R 775 /usr/local/bin/fabric-ca-client
```

第四步：检查。

```
fabric-ca-server version

fabric-ca-server version
```

显示版本信息为安装正确。

#### 2. Fabric-ca 在 MacOS 上面的安装

第一步：安装系统组件包。

```
brew install libtool
```

第二步：下载源代码并编译。

在编译之前需要打开 `$GOPATH/src/github.com/hyperledger/fabric-ca` 下面的 Makefile 文



件找到其中的第一个 GO\_LDFLAGS, 并在该行末尾加上 -s:

```
cd $GOPATH/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric-ca
cd $GOPATH/src/github.com/hyperledger/fabric-ca
make fabric-ca-server
make fabric-ca-client
```

第三步: 安装编译好的可执行文件。

```
cd $GOPATH/src/github.com/hyperledger/fabric-ca/bin
cp $GOPATH/src/github.com/hyperledger/fabric-ca/bin/* /usr/local/bin
chmod -R 775 /usr/local/bin/fabric-ca-server
chmod -R 775 /usr/local/bin/fabric-ca-client
```

第四步: 检查。

```
fabric-ca-server version
fabric-ca-server version
```

显示版本信息为安装正确。

### 6.3.2 fabric-ca-server 的启动和配置

fabric-ca-server 启动之后是以守护进程方式存在, 可以通过 fabric-ca-client 或者实现其通信协议的客户端发起请求。fabric-ca-serve 有三种方式设置配置信息, 分别是启动参数、环境变量和配置文件。

#### 1. fabric-ca-server 的命令行选项

fabric-ca-server 模块有三个子命令, 这三个子命令分别是:

- init: 初始化 fabric-ca 服务器。
- start: 启动 fabric-ca 服务器。
- version: 显示版本。

#### 2. fabric-ca-serve 的选项

fabric-ca-serve 的子命令没有各自独有的命令行选项, 所有的子命令共用一组通用的全局选项, 这些选项及其作用如下所示:

- --address: Fabric-ca 服务器的监听地址 (默认为 “0.0.0.0”)。
- -b, --boot: 系统启动对应的管理员账号和密码。
- --ca.certfile: CA 证书文件 (默认为 “ca-cert.pem”)。

- `--ca.chainfile`: CA 链证书文件 (默认为 “`ca-chain.pem`”)。
- `--ca.keyfile`: CA 密钥文件 (默认为 “`ca-key.pem`”)。
- `-n, --ca.name`: 证书颁发机构名称。
- `--cacount`: CA 实例的数量。
- `--cafiles`: 以逗号分隔的 CA 配置文件的列表。
- `--crl.expiry`: CRL 请求到期时间 (默认为 24h)。
- `--crlsizelimit`: 可接受的 CRL 的大小限制, 以字节为单位 (默认为 512000)。
- `--csr.cn`: 请求父 Fabric-ca 服务器的证书签名时使用的公用名称。
- `--csr.hosts`: 逗号分隔的父类 Fabric-ca 服务器的主机名, 支持多个。
- `--csr.serialnumber`: 请求父类 Fabric-ca 服务器的序列号。
- `--db.datasource`: 数据库的名称 (默认为 “`fabric-ca-server.db`”), 仅仅针对 `--db.type` 选项为 `sqlite3` 时有效。
- `--db.tls.certfiles`: 和数据库 TLS 通信时用的证书文件, PME 格式 (例如 `root1.pem`, `root2.pem`)。
- `--db.tls.client.certfile`: 和数据库进行 TLS 通信时客户端的证书文件, PME 格式。
- `--db.tls.client.keyfile`: 和数据库进行 TLS 通信时客户端的私钥文件, PME 格式。
- `--db.type`: 存储账号类型的数据库的类型; 目前支持三种数据库类型, 包括 `sqlite3`、`postgres`、`mysql` (默认为 “`sqlite3`”)。
- `-d, --debug`: 启用调试级别日志记录。
- `-H, --home`: Fabric-ca 服务器的主目录 (默认为当前目录)。
- `--intermediate.enrollment.label`: 操作中使用的标签。
- `--intermediate.enrollment.profile`: 发行证书时要使用的签名配置文件的名称。
- `--intermediate.parentserver.caname`: 服务器 CA 名称。
- `-u, --intermediate.parentserver.url`: 父 Fabric-ca 服务器的 URL。
- `--ldap.enabled`: 启用 LDAP 服务进行客户端身份验证和相关属性的管理。
- `--ldap.groupfilter`: LDAP 进行组过滤模式, 默认值为 (`memberUid=%s`)。
- `--ldap.tls.certfiles`: LDAP 服务器的证书文件, PEM 格式 (例如 `root1.pem`, `root2.pem`)。
- `--ldap.tls.client.certfile`: LDAP 服务客户端的证书文件, PEM 格式。
- `--ldap.tls.client.keyfile`: LDAP 服务的客户端私钥文件, PEM 格式。
- `--ldap.url`: LDAP 服务的 URL。
- `--ldap.userfilter`: LDAP 服务器的用户过滤器, 默认为 (`uid=%s`)。
- `-p, --port`: Fabric-ca 服务器监听端口 (默认值为 7054)。

- `--registry.maxenrollments` : 最大允许注册的用户数; 如果 LDAP 未启用时有效 (默认为 -1)。
- `--tls.certfile`: Fabric-ca 服务器的证书, PEM 格式 (默认 “`tls-cert.pem`”)。
- `--tls.clientauth.certfiles` : Fabric-ca 服务器的客户端证书 (例如 `root1.pem`, `root2.pem`)。
- `--tls.clientauth.type`: 客户端类型 (默认为 “`noclientcert`”)。
- `--tls.enabled`: 在监听端口上启用 TLS。

### 3. fabric-ca-server 的初始化

除了命令行参数, `fabric-ca-server` 还可以通过配置文件进行参数设置。执行 `fabric-ca-server` 的命令 `init`, 可以初始化生成相关的配置文件。在执行 `init` 命令之前需要创建相关的文件夹, 本书例子的文件夹目录是采用如下命令创建的:

```
mkdir -p /opt/hyperledger/fabric-ca-server
```

`fabric-ca-server` 服务器初始化的命令如下:

```
fabric-ca-server init -b admin:adminpw
```

`-b` 参数后面的是 `fabric-ca-server` 服务器管理账号的用户名和密码, 这里给出的是个例子, 而且很多资料都给出了这个例子, 但是如果在生产系统中建议务必使用比较复杂的密码, 而避免使用本例中的用户名和密码, 避免恶意猜测用户名和密码的行为。

`fabric-ca-server` 初始化命令执行完成之后会在当前的目录下面生成相应的配置文件, 这些文件的名字及其作用如下所示:

- `fabric-ca-server-config.yaml`: 配置文件。
- `fabric-ca-server.db`: 数据库文件 (数据库选择 `sqlite3` 时有效)。
- `ca-cert.pem`: 证书文件。
- `msp`: 私钥文件夹。

### 4. fabric-ca-server 的配置文件

`fabric-ca-server` 配置文件的内容可以参考上一步生成的文件 `fabric-ca-server-config.yaml` 的内容, 本书不在复述。这里我们重点介绍以下这些配置的组成部分和各个选项的功能。

`fabric-ca-server-config.yaml` 文件也采用了 YAML 格式, 如果你对 YAML 格式不是很熟悉, 可以先参考 YAML 相关资料。

`fabric-ca-server` 的配置文件一共分为 11 个部分, 下面将详细介绍各个部分的内容。

### (1) 通用配置部分

通用配置部分包含了系统一些公用属性，比如端口、运行模式等，具体配置信息如下所示：

```
port: 7054                // 端监听端口号
debug: false              // 是否调试
crlsizelimit: 512000
cacount:                  // 支持的 CA 数目
cafiles:                  // 相关 CA 配置文件
crl:
    expiry: 24h           // 授权证书的有效期
```

### (2) tls

tls 部分主要包含了 TLS 通信相关的配置，包括是否需要打开 TLS 通信，TLS 通信的证书和私钥等文件的路径等，具体配置信息如下所示：

```
tls:
    enabled: false         // 是否启用 TLS
    certfile: tls-cert.pem // TLS 证书文件
    keyfile:               // TLS 私钥文件
    clientauth:
        type: noclientcert // 客户端类型
        certfiles:         // 客户端证书类型
```

### (3) ca

ca 服务器属性的配置，包含发布证书的组织机构的名称和相关的证书文件路径等，具体配置信息如下所示：

```
ca:
    name:                  // CA 的名字，如果存在多个 CA 服务器，不能重复
    keyfile: ca-key.pem    // 私钥文件
    certfile: ca-cert.pem  // 证书文件
    chainfile: ca-chain.pem // 证书链文件
```

### (4) registry

registry 节点包含了客户端注册相关的信息，具体配置信息如下所示：

```
registry:

    maxenrollments: -1
    identities:       // 注册实体信息，可以有多个
        - name: admin
          pass: adminpw
          type: client
          affiliation: ""
          maxenrollments: -1
          attrs:
              hf.Registrar.Roles: "client,user,peer,validator,auditor"
              hf.Registrar.DelegateRoles: "client,user,validator,auditor"
```



```

hf.Revoker: true
hf.IntermediateCA: true
hf.GenCRL: true
hf.Registrar.Attributes: "*"

```

其中, maxenrollments 是同一个用户名和密码允许执行 enrollment 的次数, -1 为 unlimited, 0 表示不支持登记。

#### (5) db

db 部分包含了 Fabric-ca 服务器存储账号文件的数据类型的配置, Fabric-ca 服务器目前支持 sqlite3、postgres 和 mysql 三种数据库, 选择任何一种数据库在启动 Fabric-ca 服务器之前都需要安装, Fabric-ca 本身不会自动安装这些数据库引擎。db 部分的配置示例如下:

```

db:
  # 数据库类型
  type: sqlite3
  # 数据库连接方式, 不同的数据库是不同的
  datasource: fabric-ca-server.db
  # TLS 加密通信配置属性
  tls:
    enabled: false
    certfiles:
      - db-server-cert.pem
    client:
      certfile: db-client-cert.pem
      keyfile: db-client-key.pem

```

db 部分的配置中选择不同的数据库服务器配置格式是不一样的, 不同的数据库对应的配置格式是不一样的, 具体配置信息如下所示:

#### 选择 sqlite3 存储账号:

```

type: sqlite3
datasource: fabric-ca-server

```

系统启动会在启动目录下面生成数据库文件 fabric-ca-server.db, 数据库文件名由 datasource 节点设置, 该文件为 sqlite3 的数据库文件。

#### 选择 mysql 存储账号:

```

db:
  type: mysql
  datasource: root:rootpw@tcp(localhost:3306)/fabric_ca?parseTime=true&tls=custom

```

#### postgres 存储账号:

```

db:
  type: postgres
  datasource: host=localhost port=5432 user=Username password=Password dbname=fabric_
  ca sslmode=verify-full

```

如果使用 mysql 或者 postgres 存储账号, 可以选择配置采用 TLS 加密的方式和服务器进行通信。具体的配置信息如下所示:

```
tls:
  enabled: true                // 是否采用加密的方式和服务器进行通信
  certfiles:                  // TLS 加密通信的证书文件
    - db-server-cert.pem
  client:
    certfile: db-client-cert.pem // 客户端 TLS 通信证书文件
    keyfile: db-client-key.pem   // 客户端 TLS 通信私钥文件
```

#### (6) ldap

Fabric-ca 可以配置使用远端 LDAP 服务器来进行注册管理并且保存注册相关的数据, LDAP 服务相关的配置信息包含在 ldap 节点中, 具体配置信息如下所示:

```
ldap:
  enabled: true
  url: ldap://cn=admin,dc=example,dc=org:admin@localhost:10389/dc=example,dc=org
  userfilter: (uid=%s)
  tls:
    certfiles:
      - ldap-server-cert.pem
    client:
      certfile: ldap-client-cert.pem
      keyfile: ldap-client-key.pem
```

#### (7) affiliations

affiliations 节点包含了组织中的部门的相关配置信息, 这些配置信息在客户端 SDK 调用时相关的参数必须保持一致。affiliations 配置信息如下所示:

```
affiliations:
  org1: // 组织 org1 中的部门
    - department1
    - department2
  org2: // 组织 org2 中的部门
    - department1
```

在通过 Fabric 的客户端 SDK 访问 Fabric-ca 服务器的时候, 相关的方法需要设置 affiliations 的参数时, 需要跟这里的配置保持一致, 否则无法正确访问。具体可以参考本书第 9 章相关的内容。

#### (8) signing 节点

signing 节点包含了证书签发相关的配置, 包括证书的到期时间等属性。signing 相关的

配置信息如下所示:

```
signing:
  default:
    // 默认的签发 Ecert
    usage:
    // 证书签发的作用域
    - digital signature
    expiry: 8760h
    // 证书有效时间
  profiles:
    ca:
    // 本节点作为父节点给下层节点签发时的模板
    usage:
    - cert sign
    expiry: 43800h
    caconstraint:
      isca: true
      maxpathlen: 0
    // 现在中间层继续下发节点
    tls:
    // TLS 通信相关配置
    usage:
    - signing
    - key encipherment
    - server auth
    - client auth
    - key agreement
    expiry: 8760h
```

#### (9) csr

csr 节点包含了证书申请请求时需要使用的配置信息。如果当前 CA 服务器是作为根 CA 服务器存在的, 那么需要设置这些属性。csr 相关的配置信息如下所示:

```
csr:
  cn: fabric-ca-server
  // 服务器名称
  names:
  // 证书签发单位的基本信息
  - C: US
    ST: "North Carolina"
    L:
    O: Hyperledger
    OU: Fabric
  hosts:
  - robertfeng-All-Series
  - localhost
  ca:
    expiry: 131400h
    // 证书有效时间
    pathlength: 1
    // 下一级服务器是否可以继续其下级服务签发证书, -1 为不可,
    // 取值大于 0 时为允许的层级数。
```

#### (10) bccsp 节点

bccsp 节点包含了加密算法相关的配置, 在 bccsp 节点中可以选择相关的加密算法以及相关加密算法的证书文件。bccsp 节点的配置信息如下所示:

```
bccsp:
```

```

default: SW           // 加密方式，硬件加密和程序加密
sw:
  hash: SHA2
  security: 256
  filekeystore:
    keystore: msp/keystore

```

### (11) intermediate

当前 CA 作为中间层时相关的配置。如果当前 CA 服务器需要从上级 CA 服务器获取授权才能工作，需要配置 intermediate 节点的相关属性。intermediate 节点的配置信息如下所示：

```

intermediate:
  parentserver:      // 上级 CA 服务器相关信息
    url:
    caname:

  enrollment:        // 需要在上级 CA 服务器进行登记的信息
    hosts:            // 上级 CA 服务器地址列表，用逗号分隔
    profile:          // 签发用的 profile
    label:            // 用于 HSM 操作的标签信息

  tls:               // TLS 通信相关的属性信息
    certfiles:        // 根证书文件
    client:           // 如果启用客户端文件需要的使用到的部分
      certfile:       // 客户端证书文件
      keyfile:        // 客户端私钥文件

```

## 5. fabric-ca-server 的启动

配置文件设置好之后就可以启动 fabric-ca-server 服务器了，启动命令如下：

```
fabric-ca-server start -H /opt/hyperledger/fabric-ca --boot admin:adminpw
```

启动完成后可以通过客户端程序或者通过 fabric-ca-client 模块访问 fabric-ca-server 服务器。

### 6.3.3 fabric-ca-client 的使用

fabric-ca-server 提供了一组 RESTAPI 接口供第三方应用程序调用。fabric-ca-client 对这些 RESTAPI 接口进行了封装，只需设置简单的参数便可以完成账号注册、账号授权等操作。fabric-ca-client 模块由一组子命令和相关的参数选项组成。

#### 1. fabric-ca-client 模块的子命令

fabric-ca-client 模块通过相关的子命令完成账号注册、授权申请、证书撤销等操作。fabric-ca-client 模块包含的子命令如下所示：

- enroll：登记账号。



- `gencrl`: 撤销证书。
- `gencsr`: 创建证书签名。
- `getcacert` 获取 CA 链证书。
- `reenroll`: 重新登记账号。
- `register`: 注册一个新账号。
- `revoke`: 撤销一个账号。
- `version`: 显示版本信息。

## 2. fabric-ca-client 模块的参数选项

`fabric-ca-client` 模块和 `fabric-ca-server` 模块一样, 所有命令公用一组全局选项, 不同的子命令需要的选项是不一样的。`fabric-ca-client` 模块的选项及其选项的作用如下所示:

### (1) 基本管理

- `--caname`: CA 服务器名称。
- `-H, --home`: 客户端的目录, 用来存放客户端相关的文件(默认值为 `"/root/.fabric-ca-client"`)。
- `-M, --mspdir`: 客户端的账号证书文件的目录(default `"msp"`)。
- `-d, --debug`: 将客户端的日志设置为 debug 模式。
- `-u, --url`: `fabric-ca-server` 的地址(默认值为 `"http://localhost:7054"`)。

### (2) 账号登记

- `--enrollment.attrs`: 账号登记请求中的属性(e.g. `foo, bar:opt`)。
- `--enrollment.label`: 登记请求中 HSM 相关的标签。
- `--enrollment.profile`: 登记请求中的 profile。

### (3) 账号注册

- `--id.affiliation`: 账号注册时组织的部门信息。
- `--id.attrs`: 账号注册时的属性列表(e.g. `foo=foo1, bar=bar1`)。
- `--id.maxenrollments`: 当前注册后可以登记的次数(default `-1`)。
- `--id.name`: 账号注册时的用户名。
- `--id.secret`: 账号注册的密码。
- `--id.type`: 账号注册的类型, 目前支持(`'peer, app, user'`), 默认值为 `"user"`。

### (4) 证书吊销

- `-a, --revoke.aki`: 注销证书时需要的公钥。
- `-e, --revoke.name`: 注销证书时的实体名称。
- `-r, --revoke.reason`: 注销的原因。
- `-s, --revoke.serial`: 注销证书的序列号。

### (5) CA 证签名

- `--csr.cn`: CA 签名请求时, 请求报文中的通用名。
- `--csr.hosts`: CA 签名请求时, 请求报文中的主机名。
- `--csr.names`: CA 签名请求时, 请求报文中的附加名称 (比如 `C=CA, O=Org1`)。
- `--csr.serialnumber`: CA 签名请求时, 请求报文中的序列号。
- `-m, --myhost`: CA 签名请求时, 请求报文中请求主机名称, 默认值是本地主机。

### (6) TLS 通信

- `--tls.certfiles`: TLS 通信模式下的证书文件, PEM 文件格式。
- `--tls.client.certfile`: TLS 通信模式下客户端的证书文件, PEM 文件格式。
- `--tls.client.keyfile`: TLS 通信模式下客户端的私钥文件。

## 3. fabric-ca-client 常用的命令

### (1) 注册新账号

```
fabric-ca-client register --id.name peer2 --id.type peer --id.affiliation
org1.department1 --id.secret peer2wd -u http://localhost:7054
```

### (2) 载入账号信息

```
fabric-ca-client enroll -M /usr/fabric-test1/msp -u http://peer1:peer1pw@
localhost:7054
```

### (3) 获取 CA 服务器的证书

```
fabric-ca-client getcacert -u http://localhost:7054 -M /usr/fabric-test1/msp
```

## 6.4 将 fabric-ca-server 绑定到现有项目中

Fabric-ca 是 Fabric 的 cryptogen 模块的有力补充, 在实际项目中占有非常重要的作用。在 Fabric 项目一般首先采用 cryptogen 模块生成组织、Peer 节点、Orderer 节点等模块的账号文件, 但是如果项目中需要动态生成用户账号文件, 这个时候 Fabric-ca 能提供更好的帮助。为了让 Fabric-ca 能够动态地为指定的组织生成用户账号, 需要将 Fabric-ca 和相关的组织进行绑定。

### 1. 绑定 fabric-ca-server 到现有组织

将 fabric-ca-server 绑定到某个组织的操作还是比较简单的。下面的示例中我们将演示如何将按照 6.3.2 节的过程启动的 fabric-ca-server 服务器绑定到 5.2.1 节示例中生成的组织 org1 中。首先打开 fabric-ca-server 的配置文件 `fabric-ca-server-config.yaml`, 在配置文件中找到以下内容:

```
ca:
  name: ca-org1
  keyfile:
  certfile:
  chainfile: ca-chain.pem
```

然后进入 5.2.1 节中生成的证书文件夹, 进入存放组织 org1 相关的证书的文件夹, 执行命令 `tree -L 2`:

```
.
├── ca
│   ├── a1db721c0cfb6f107fc45501d29866633d21a3492c87bb352687b2e8e85b652e_sk
│   ├── ca-key.pem
│   └── ca.org1.qklszzn.com-cert.pem
├── msp
│   ├── admincerts
│   ├── cacerts
│   └── tlscacerts
├── peers
│   ├── peer0.org1.qklszzn.com
│   ├── peer1.org1.qklszzn.com
│   ├── peer2.org1.qklszzn.com
│   └── peer3.org1.qklszzn.com
├── tlsca
│   ├── f92d78e8195795e3a115c4c129d78edaac70ec4f5ef4764df43e1de245e7d467_sk
│   └── tlsca.org1.qklszzn.com-cert.pem
└── users
    ├── Admin@org1.qklszzn.com
    ├── User1@org1.qklszzn.com
    ├── User2@org1.qklszzn.com
    ├── User3@org1.qklszzn.com
    ├── User4@org1.qklszzn.com
    ├── User5@org1.qklszzn.com
    └── User6@org1.qklszzn.com
```

在 ca 文件夹中存放了将 fabric-ca-server 绑定到组织 org1 的相关文件。

读者 ca 文件夹中文件名可能和这里的不一样, 这些都是正常的。

现在可以绑定这些文件了, 绑定之后的 fabric-ca-server 的配置文件 fabric-ca-server-config.yaml 中相关的配置信息如下:

```
ca:
  name: ca-org1
  keyfile: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/org1.qklszzn.com/ca/a1db721c0cfb6f107fc45501d29866633d21a3492c87bb352687b2e8e85b652e_sk
  certfile: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/org1.qklszzn.com/ca/ca.org1.qklszzn.com-cert.pem
  chainfile: ca-chain.pem
```

- keyfile: 对应 ca 文件夹中文件名后缀为 \_sk 的文件。
- certfile: 对应 ca 文件夹中文件名为 ca.org1.qklszzn.com-cert.pem 的文件。

通过上述步骤 fabric-ca-server 就被绑定到组织 org1 中了。

## 2. 通过客户端从已经绑定的 fabric-ca-server 中生成账号

现在我们通过一个例子来演示如何通过 fabric-ca-client 从已经绑定到指定组织中的 fabric-ca-server 中获取一个新的用户账号。

第一步: 设置 fabric-ca-client 环境变量。

fabric-ca-client 在使用之前需要创建一个目录, 该目录存放 fabric-ca-client 的账号 (msp) 文件, 本例中使用的目录如下所示:

```
mkdir -p /opt/hyperledger/fabric-client
```

目录创建完成之后, 利用管理员账号和密码注册, 获取管理员账号 (msp) 的证书文件。

```
export FABRIC_CA_CLIENT_HOME=/opt/hyperledger/fabric-client
fabric-ca-client enroll -u http://admin:adminpw@localhost:7054 -M
/opt/hyperledger/fabric-client
```

第二步: 注册账号。

现在开始注册一个用户名为 usertest, 密码为 user2wd 的账号, 注册命令示例如下所示:

```
fabric-ca-client register --id.name usertest --id.type user --id.affiliation
org1.department1 --id.secret user2wd -u http://localhost:7054
```

第三步: 载入账号。

现在将上一步注册的账号 usertest 加载到本地, 首先需要在本地创建要给目录用来存放从服务器下载的证书, 目录可以是任何目录。本例的目录如下所示:

```
mkdir -p /opt/hyperledger/qklszznuser
```

在上述命令创建的目录中, 登记账号 usertest, 并将登记成功的账号的相关文件保存到指定目录中。登记账号的命令如下所示:

```
fabric-ca-client enroll -u http://usertest:user2wd@localhost:7054 -M /opt/
hyperledger/qklszznuser/msp
```

第四步: 复制管理员签名和公用 TLS 证书文件。

复制管理账号的签名的命令如下所示:

```
mkdir /opt/hyperledger/qklszznuser/msp/admincerts
cp /opt/hyperledger_commnconfig/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp/signcerts/* /opt/hyperledger/
qklszznuser/msp/admincerts
```



复制公用 TLS 签名证书的命令如下所示:

```
mkdir /opt/hyperledger/qklszznuser/tls
cp /opt/hyperledger_commnconfig/fabricconfig/crypto-config/peerOrganizations/
org1.qklszzn.com/peers/peer0.org1.qklszzn.com/tls/* /opt/hyperledger/qklszznuser/
tls
```

第五步: 查看账号。

进入账号文件夹通过 `tree` 命令查看证书文件的命令, 如下所示:

```
.
├── msp
│   ├── admincerts
│   │   └── Admin@org1.qklszzn.com-cert.pem
│   ├── cacerts
│   │   └── 192-168-23-212-7054.pem
│   ├── keystore
│   │   └── 5f470b06e7b34517e1f5bc6b105e2c9ec4a47759378997e826de6f305c075b47_sk
│   ├── signcerts
│   │   └── cert.pem
│   └── tlscacerts
│       └── tls-192-168-23-212-7054.pem
└── tls
    ├── ca.crt
    ├── server.crt
    └── server.key
```

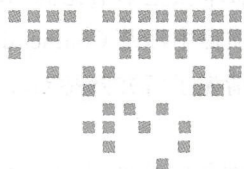
通过观察文件结构我们发现和 `cryptogen` 模块生成的账号文件是一致的, 只是文件名不一样, 这是正常的。

### 3. fabric-ca-server 客户端访问接口

在 Fabric 项目中更多的应用场景是客户端程序通过 `fabric-ca-server` 提供的 RESTAPI 接口完成账号注册、账号登记等操作。在本书的第 8 章中将会详细介绍第三方应用程序如何通过相应的 SDK 访问 `fabric-ca-server` 服务器。

## 6.5 本章小结

本章着重介绍了 Fabric 账号体系, 通过阅读本章内容可以熟悉并了解 Fabric 是如何通过账号来进行权限控制的。灵活运用 Fabric 提供的账号功能是开发基于 Fabric 技术框架项目的先决条件。



## Fabric 的智能合约详解

智能合约是区块链里面一个非常重要的概念和组成部分。在 Fabric 中将智能合约称为 Chaincode，中文翻译为链码。注意，在本章以及本章后面的章节中，凡是涉及链码的地方我们统称为 Chaincode。本章将会详细地介绍 Chaincode 的基本概念、代码结构、开发和调试方法以及部署等过程。通过本章的阅读，读者可以理解并且能在项目中灵活熟练地使用 Fabric 的 Chaincode。

### 7.1 Chaincode 初探

Fabric 中的 Chaincode 包含了 Chaincode 代码和 Chaincode 管理命令这两部分的内容。其中：

- Chaincode 代码是业务的承载体，负责具体的业务逻辑；
- Chaincode 管理命令负责 Chaincode 的部署、安装、维护等工作。

#### 1. Chaincode 代码

Fabric 的 Chaincode 是一段运行在容器中的程序，这些程序可以是 Go、Java、Node.js 等语言开发的。Chaincode 是客户端程序和 Fabric 之间的桥梁。通过 Chaincode 客户端程序可以发起交易、查询交易。Chaincode 是运行在 Docker 容器中的，因此 Chaincode 相对比较安全。在 Chaincode 开发语言的支持中，目前 Fabric 支持使用 Golang、Java 和 Node.js 这三种编程语言开发 Chaincode。在这些开发语言中，Golang 是比较成熟稳定的，Java 和 Node.js

这两个语言的版本截至本书完稿的时候还在完善中。因此,本书中所有和 Chaincode 相关的实例以及演示项目中涉及 Chaincode 的地方,一律采用 Golang 开发。如果读者对 Golang 不是非常熟悉,请先了解并熟悉 Golang 的语法结构和技术特性之后再阅读本章内容,效果会更好。

如果读者准备阅读的过程中进行同步练习,请先按照本书第 4 章的内容搭建好一个简单的 Fabric 环境,在这个环境中只需要包含一个 orderer 节点和一个 peer 节点就可以了。

## 2. Chaincode 管理命令

Chaincode 管理命令主要用来对 Chaincode 进行安装、实例化、调用、打包、签名等操作。

Chaincode 命令包含在 peer 模块中,是 peer 模块的一个子命令,该子命令的名称为 chaincode,该子命令的格式如下所示:

```
peer chaincode
```

在下面的章节中,将会详细介绍 peer 模块的 Chaincode 子命令及其参数选项的作用。

## 7.2 快速编写和运行一个 Chaincode

Golang 是目前开发 Chaincode 最成熟最稳定的语言,本节我们将介绍如何使用 Golang 开发一个简单的 Chaincode。

首先我们将快速地创建一个简单的 chaincode,让各位读者对 Chaincode 的代码结构及其安装、部署、调用的方式有一个直观的概念。

### 1. 创建 Chaincode 代码的目录

首先需要创建一个目录存放 Chaincode 的代码,Chaincode 代码的目录可以是任何路径,但是为了查询方便,建议将 Chaincode 存放在 \$GOPATH 指定的路径中。本例创建 Chaincode 代码的路径的命令如下所示:

```
mkdir -p $GOPATH/src/github.com/qklszzl/chaincodestudy/simpledemo
```

### 2. 创建 Chaincode 源代码文件并编写源代码

创建域代码文件的命令如下所示:

```
vi simplechaincode.go
```

在源代码文件 simplechaincode.go 中输入以下代码:

```
package main  
  
import (
```

```

    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"

)

// 定义一个机构体, 作为 chaincode 的主对象, 可以是任何符合 Go 语言规范的命名方式
type simplechaincode struct {

}

/**
    系统初始化方法, 在部署 chaincode 的过程中当执行命令的时候会调用该方法

    peer chaincode instantiate -o orderer.qklszzn.com:7050 -C
        qklszzlchannel -n r_test_cc6 -v 1.0 -c '{"Args":["init","a","100","b","200"]}'
        -P "OR      ('Org1MSP.member','Org2MSP.member')"

*/
func (t *simplechaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

    fmt.Println(" <<  =====  success init it is view in docker  =====
>> ")

    return shim.Success([]byte("success init "))
}

/**

    主业务逻辑, 在执行命令的时候系统会调用该方法并传入相关的参数, 注意 "invoke" 之后的参数是需
    要传入的参数

    peer chaincode invoke -o 192.168.23.212:7050 -C qklszzlchannel -n r_test_cc6
    -c '{"Args":["invoke","a","b","1"]}'

*/
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    fmt.Println("  =====  1、success it is view in docker  ===== ")
    return shim.Success([]byte("success invok "))

}

func main() {
    err := shim.Start(new(simplechaincode))
    if err != nil {

```



```

    fmt.Printf("Error starting Simple chaincode: %s", err)
}
}

```

至此一个简单的 Chaincode 代码就完成了。接下来我们会将已经完成的 Chaincode 的源代码部署到 Fabric 中。Fabric 是通过 peer 模块的子命令 chaincode 来完成 Chaincode 的部署、发布、更新等操作的。在 Chaincode 安装之前首先要确保已经按照本书第 4 章的步骤成功编译安装了 Fabric 的相关子模块, 并且要保证已经正确启动 Orderer 节点和 Peer 节点。chaincode 的部署涉及 Channel, 请读者先严格按照本书第 5 章的内容创建一个 channel。如果读者修改了一些参数, 请在执行下面命令的时候注意 -C 参数后面的值, 这个值是表示 channel 的名称。

在部署之前首先通过下面的命令查看当前的 Peer 节点已经加入了那些 Channel。

```

export set FABRIC_CFG_PATH=/opt/hyperledger/peer
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszsn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszsn.com/users/Admin@org1.qklszsn.com/msp

peer channel list

```

上述命令会显示当前的 Peer 节点所加入的 Channel, 下面步骤中 -C 选项后的值必须在上述命令执行的结果中。下面开始部署并且调用 Chaincode。

第一步: 部署。

```

export set FABRIC_CFG_PATH=/opt/hyperledger/peer
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszsn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszsn.com/users/Admin@org1.qklszsn.com/msp

peer chaincode install -n qklszsncc -v 1.1 -p github.com/qklszsl/chaincodestudy/
simpledemo

```

第二步: 实例化。

```

export set FABRIC_CFG_PATH=/opt/hyperledger/peer
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszsn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszsn.com/users/Admin@org1.qklszsn.com/msp

peer chaincode instantiate -o orderer.qklszsn.com:7050 -C qklszslchannel -n
qklszsncc -v 1.1 -c '{"Args":["init","a","100","b","200"]}' -P "OR
('Org1MSP.member','Org2MSP.member')"

```

Chaincode 实例化成功之后运行 Chaincode 的 Docker 容器会被启动, 这个时候执行命令 `dokcer ps`, 会发现一个包含 chanincode 名字的进程以及被运行。Chaincode 如果被成功

实例化, 在当前的 Peer 节点被重新启动之后, 已经被实例化的 Chaincode 不会自动重新启动, 这个时候如果客户端对 Chaincode 发起请求 (比如请求 query 方法), 系统会自动运行 Chaincode 的 Docker 进程。还有一点需要注意, 如果 Chaincode 某个方法发生异常导致 Docker 容器关闭, 若此时客户端重新对 Chaincode 发起访问请求, 只要请求的方法没有异常, 系统会自动启动 Chaincode 的容器。

关于 Chaincode 的示例还有一点需要注意, 如果在一个 channel 中已经加入了多个 peer 节点, 并且这些 peer 节点需要安装相同的 chaincode。这个时候只需要在第一个部署 Chaincode 的 Peer 节点中执行 peer chaincode instantiate, 其余的 Peer 节点部署 Chaincode 的时候只需要执行 peer chaincode install 命令, 然后执行 invoke 或者 query 命令, 系统会自动启动 Chaincode 相关的 Docker 镜像。当需要在多个 Peer 节点部署同一个 Chaincode 的时候 instantiate 命令只需要执行一次。

第三步: 调用。

```
export set FABRIC_CFG_PATH=/opt/hyperledger/peer
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer chaincode invoke -o orderer.qklszzn.com:7050 -C qklszzlchannel -n
qklszzncc -c '{"Args":["invoke","1","a","b"]}'
```

如果没有出现错误那我们就完成了一个最简单的 chaincode 的代码编写、部署、发布、调用的过程。

## 7.3 Golang 版本的 Chaincode 的代码结构

通过本章前面的内容, 我们已经了解了 chaincode 是如何部署到 Fabric 中。现在我们剖析以下 Chaincode 代码的结构, 这里以 Golang 版本的 Chaincode 源代码为例。

### 7.3.1 Chaincode 源代码的基本结构

#### 1. 包名

一个 chaincode 通常是一个 Golang 源代码文件, 在这份源代码中, 包名必须是 main。

```
package main
```

#### 2. 引入包

Chaincode 需要引入 Fabric 提供的一些系统包, 这些系统提供了 Chaincode 和 Fabric 进

行通信的接口。引入包的源代码如下所示:

```
"fmt"
"github.com/hyperledger/fabric/core/chaincode/shim"
pb "github.com/hyperledger/fabric/protos/peer"
```

在引入包中, `github.com/hyperledger/fabric/core/chaincode/shim` (在下文中我们简称为 `shim`) 是 Fabric 系统提供的上下文环境, 包含了 Chaincode 和 Fabric 交互的接口。在 Chaincode 中, 执行赋值、查询等功能都需要通过 `shim`。



**注意** `fmt` 是 Golang 系统提供的通用输入输出包, 不是必需的。但是上例中 `fmt` 包后面的两个包都是必须引入的。

### 3. 定义结构体并实现

每个 chaincode 都需要定义一个结构体, 结构体的名字可以是任意符合 Golang 命名规范的字符串。

```
type chainCodeStudy1 struct {
}
```

chaincode 结构体是 chaincode 的主体结构。chaincode 结构体需要实现 Fabric 提供的接口 `"github.com/hyperledger/fabric/protos/peer"`, 其中必须需要实现下面两个方法:

```
func (t *chainCodeStudy1) Init(stub shim.ChaincodeStubInterface) pb.Response {}
func (t *chainCodeStudy1) Invoke(stub shim.ChaincodeStubInterface) pb.Response {}
```

### 4. Chaincode 的 Init 方法

Init 方法是系统初始化方法, 当执行命令 `peer chaincode instantiate` 实例化 chaincode 的时候会调用该方法, 同时命令中 `-c` 选项后面内容会作为参数传入 Init 方法中。以下面的 chaincode 实例化命令为例:

```
peer chaincode instantiate -o orderer.qklszzn.com:7050 -C qklszzlchannel -n
sqlkszzncc -v 1.0 -c '{"Args":["init","a","100","b","200"]}'
```

上面命令给 Chaincode 传入 4 个参数 “a”、“100”、“b”、“200”。注意命令中 `Args` 后面一共有 5 个参数, 其中第一个参数 `init` 是固定值, 后面的才是参数。传参数的个数是没有限制的, 但是实际应用的时候不要太多。如果有很多参数需要传递给 chaincode, 可以采用一些数据格式 (比如 JSON), 把数据格式化之后传给 Chaincode。在 Init 方法中可以通过下列方法获取传入参数。

```
func (t *chainCodeStudy1) Init(stub shim.ChaincodeStubInterface) pb.Response {
    // 获取客户端传入的参数，args 是一个字符串数据，存储传入的字符串参数
    _, args := stub.GetFunctionAndParameters()
}
```

## 5. Invoke 方法

Invoke 方法的主要作用是写入数据，比如发起交易等。在执行命令 `peer chaincode invoke` 的时候系统会调用该方法，同时会把命令中 `-c` 后面的参数传入 Invoke 方法中。下面的 `invoke` 命令为例：

```
peer chaincode invoke -o 192.168.23.212:7050 -C qklszzlchannel -n qlkszzncc -c
'{"Args":["invoke","a","b","1"]}'
```

上面的命令调用 Chaincode 的 Invoke 方法并且传入三个参数 “a” “b” “1”。注意 Args 后面数组中的第一个值 “invoke” 是默认的固定参数。

### 7.3.2 shim 包的核心方法

在 Fabric 的 Golang 语言的 Chaincode 源代码中如需要引入系统包 `"github.com/hyperledger/fabric/core/chaincode/shim"` (在本章中简称 shim)。shim 包主要负责和客户端进行通信。shim 提供了一组核心方法和客户端进行交互，这些方法如下所示。

#### 1. Success

Success 方法负责将正确的消息返回给调用 Chaincode 的客户端，Success 方法的定义和调用示例如下：

```
// 方法定义
func Success(payload []byte) pb.Response

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    return shim.Success( []byte( "success invok " ) )
}
```

#### 2. Error

Error 方法负责将错误的信息返回给调用 Chaincode 的客户端，Error 方法的定义和调用示例如下：

```
// 方法定义
func Error(msg string) pb.Response

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
```



```

    shim.Error(error_str)
}

```

### 3. LogLevel

LogLevel 方法负责修改 Chaincode 中运行日志的级别，LogLevel 方法的定义和调用示例如下：

```

// 方法定义
func LogLevel(levelString string) (LoggingLevel, error)

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    logleve , _ := shim.LogLevel("debug")
    shim.SetLoggingLevel(logleve)
    return shim.Success([]byte("success invok  and Not opter !!!!!!! "))
}

```

### 7.3.3 ChaincodeStubInterface 接口中的核心方法

在 shim 包中有一个接口 ChaincodeStubInterface，在 Invoke 方法和 Query 方法中该接口作为参数传入方法中。ChaincodeStubInterface 接口提供了一组方法，通过这组方法可以非常方便地操作 Fabric 中的账本数据。ChaincodeStubInterface 接口的核心方法大概可以分四个大类：系统管理、存储管理、交易管理、调动外部 chaincode。这些方法如表 7-1 所示。

表 7-1 Chaincode shim 包的方法名称和功能对应表

方法名称	功 能
PutState	存储数据到账本中
DelState	删除账本中的数据
GetState	从账本中获取指定的数据
CreateCompositeKey	创建复合键
GetStateByPartialCompositeKey	通过复合键取值
SplitCompositeKey	拆分复合键
GetStateByRange	查询指定 key 指定范围的数据
GetHistoryForKey	获取指定 key 的历史记录
GetTxID	获取交易编号
GetTxTimestamp	获取交易的时间
GetCreator	获取交易创建者的时间
InvokeChaincode	调用其他的 Chaincode

下面我们将通过具体的示例来介绍这些方法的定义和调用方式。

## 1. ChaincodeStubInterface 接口的系统管理相关方法

**GetFunctionAndParameters**：该方法负责接收调用 Chaincode 的客户端传递过来的参数，它的定义和调用示例如下：

```
// 方法定义
GetFunctionAndParameters() (string, []string)

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    _, args := stub.GetFunctionAndParameters()
    var a_parm = args[0]
    var b_parm = args[1]
    var c_parm = args[2]

}
```

这里以一个命令调用 Chaincode 的示例来说明 GetFunctionAndParameters 方法是如何接收客户端传递过来的参数的。示例命令的格式如下：

```
peer chaincode invoke -o 192.168.23.212:7050 -C qklszzlchannel -n qklszzncc -c
'{"Args":["invoke","set","akeym","11234343"]}'
```

上述命令 GetFunctionAndParameters 方法中将会接收到来自客户端的三个参数，-c 选择后面就是传递给客户端的参数。

根据 Invoke 子命令的格式定义，Args 不是参数而是格式关键字，后面的参数数组中第一个是方法名，后面三个是参数。

## 2. ChaincodeStubInterface 接口的存储管理相关方法

### (1) PutState

**PutState** 方法负责把客户端传递过来的数据保存到 Fabric 中，它的定义和调用示例如下：

```
// 方法定义
PutState(key string, value []byte) error

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    stub.PutState(b_parm, []byte("putvalue"))
    return shim.Success( []byte( "success invok " + c_parm ) )
}
```

### (2) GetState

**GetState** 方法负责从 Fabric 中取出数据，然后把这些数据交给 Chaincode 处理，它的定

义和调用示例如下:

// 方法定义

```
GetState(key string) ([]byte, error)
```

// 代码示例

```
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    var keyvalue []byte
    var err error
    keyvalue, err = stub.GetState("getkey")

    if( err != nil ){

        return shim.Error(" find error! ")
    }

    return shim.Success( keyvalue )
}
```

### (3) GetStateByRange

GetStateByRange 方法根据 key 的范围来查询相关的数据。它的定义和调用示例如下:

// 方法定义

```
GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)
```

// 代码示例

```
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    startkey := "startKey"
    endkey := "endkey"

    keysIter, err := stub.GetStateByRange( startkey , endkey )

    if err != nil {
        return shim.Error(fmt.Sprintf(" GetStateByRange find err : %s",
err))
    }

    defer keysIter.Close()

    var keys []string

    for keysIter.HasNext(){

        response, iterErr := keysIter.Next()

        if iterErr != nil{
```

```

        return shim.Error(fmt.Sprintf("find an error %s", iterErr))
    }

    keys = append(keys, response.Key)
}

for key, value := range keys {
    fmt.Printf("key %d contains %s\n", key, value)
}

jsonKeys, err := json.Marshal(keys)
if err != nil {
    return shim.Error(fmt.Sprintf(" find error on marshaling JSON:
%s", err))
}

return shim.Success(jsonKeys)
}

```

#### (4) GetHistoryForKey

GetHistoryForKey 方法负责查询某个键的历史记录，它的定义和调用示例如下：

// 方法定义

```
GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error)
```

// 代码示例

```

func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    keysIter, err := stub.GetHistoryForKey(b_parm);

    if err != nil {
        return shim.Error(fmt.Sprintf("GetHistoryForKey failed. Error accessing
state: %s", err))
    }
    defer keysIter.Close()

    var keys []string

    for keysIter.HasNext() {

        response, iterErr := keysIter.Next()
        if iterErr != nil {
            return shim.Error(fmt.Sprintf("GetHistoryForKey operation failed.
Error accessing state: %s", err))
        }
    }
}

```



```

// 交易编号
txid := response.TxId
// 交易的值
txvalue := response.Value
// 当前交易的状态
txstatus := response.IsDelete
// 交易发生的时间戳
txtimesamp := response.Timestamp

tm := time.Unix(txtimesamp.Seconds, 0)
datestr := tm.Format("2006-01-02 03:04:05 PM")

fmt.Printf(" Tx info - txid : %s value : %s if delete: %t datetime
: %s \n ", txid , string(txvalue) , txstatus , datestr )

keys = append( keys , txid)

}

jsonKeys, err := json.Marshal(keys)
if err != nil {
    return shim.Error(fmt.Sprintf(" Query operation failed. Error marshaling
JSON: %s", err))
}

return shim.Success(jsonKeys)
}

```

### (5) DelState

DelState 方法用来删除一个 key，它的定义和调用示例如下：

// 方法定义

DelState(key string) error

// 代码示例

```

func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    err := stub.DelState( "deletkey" )
    if err != nil {
        return shim.Error(" delete error !!!!! ")
    }
    return shim.Success([]byte(" delete success !!!!! "))
}

```

### (6) CreateCompositeKey

CreateCompositeKey 方法负责创建一个组合键，它的定义和调用示例如下：

// 方法定义

```
CreateCompositeKey(objectType string, attributes []string) (string, error)
```

// 代码示例

```
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    parms := []string{ "c_1" , "d_1" , "e_1","f_1","g_1","h_1" }
    ckey ,_ := stub.CreateCompositeKey( "testkey" , parms )

    err := stub.PutState(ckey , []byte(c_parm) )

    if err !=nil{

        fmt.Println(" find errors  %s", err)
    }

    return shim.Success([]byte(ckey))

}
```

### (7) GetStateByPartialCompositeKey

GetStateByPartialCompositeKey 方法用来查询复合键的值，它的定义和调用示例如下：

// 方法定义

```
GetStateByPartialCompositeKey(objectType string, keys []string) (StateQueryIteratorInterface,
error)
```

// 代码示例

```
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    parms := []string{ "c_1" }
    rs, err := stub.GetStateByPartialCompositeKey( "testkey" , searchparm )

    if err != nil {
        error_str := fmt.Sprintf("find error: %s", err)
        return shim.Error(error_str)
    }

    defer rs.Close()

    var i int
    var tlist []string
    for i = 0; rs.HasNext(); i++ {

        responseRange, err := rs.Next()
```

```

        if err != nil {
            error_str := fmt.Sprintf(" find error:  %s", err)
            fmt.Println(error_str)
            return shim.Error(error_str)
        }

        value1, compositeKeyParts, _ := stub.SplitCompositeKey(responseRange.
Key)

        value2 := compositeKeyParts[0]
        value3 := compositeKeyParts[1]

        fmt.Printf(" find value v1:%s v2:%s v3:%s\n", value1, value2, value3)

    }

    return shim.Success("success")
}

```

### (8) SplitCompositeKey

SplitCompositeKey 方法用来拆分复合键的属性，它的定义示例如下：

// 方法定义

```
SplitCompositeKey(compositeKey string) (string, []string, error)
```

SplitCompositeKey 方法的调用示例请参考 GetStateByPartialCompositeKey 方法的调用示例。

### 3. ChaincodeStubInterface 接口中交易管理相关的方法

GetTxTimestamp：该方法负责获取当前客户端发送的交易时间戳，它的定义和调用示例如下：

// 方法定义

```
GetTxTimestamp() (*timestamp.Timestamp, error)
```

// 代码示例

```

func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    txtime,err:= stub.GetTxTimestamp()
    if err != nil {
        fmt.Printf("Error getting transaction timestamp: %s", err)
        return shim.Error(fmt.Sprintf("Error getting transaction timestamp:
%s", err))
    }

    tm := time.Unix(txtime.Seconds, 0)

```

```

    fmt.Printf("Transaction Time: %v \n ", tm.Format("2006-01-02 03:04:05 PM"))

    return shim.Success([]byte(fmt.Sprintf(" time is : %s ",tm.Format("2006-01-02
15:04:05"))))
}

```

#### 4. 调用其他 Chaincode 的方法

##### • InvokeChaincode

在 ChaincodeStubInterface 接口中还有一个接口可以调用其他的 Chaincode，这个方法就是 InvokeChaincode。InvokeChaincode 方法的定义和调用示例如下所示：

```

// 方法定义
InvokeChaincode(chaincodeName string, args [][]byte, channel string)
pb.Response

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    // 设置参数
    parms1 := []string{"query","a"}
    queryArgs := make([][]byte, len(parms1))
    for i, arg := range parms1 {
        queryArgs[i] = []byte(arg)
    }

    // 调用 Chaincode
    response := stub.InvokeChaincode("cc_endfinlshed",queryArgs,"roberttes-
tchannel12")

    if response.Status != shim.OK {
        errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s",
response.Payload)
        fmt.Printf(errStr)
        return shim.Error(errStr)
    }

    result := string(response.Payload)

    fmt.Printf(" invoke chaincode %s ",result)

    return shim.Success([]byte("success InvokeChaincode and Not opter !!!!!!!!
" + result))

}

```

##### • GetTxID

GetTxID：该方法可以获取客户端发送的交易的编号，GetTxID 方法的定义和调用示例



如下:

```
// 方法定义
GetTxID() string

// 代码示例
func (t *simplechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    txid := stub.GetTxID();
    fmt.Println(" ===== GetTxID ===== %s ",txid)
    return shim.Success([]byte(txid))
}
```

## 7.4 Chaincode 相关的操作命令和选项

在 7.2 节中通过一个简短的例子让读者了解到了 Chaincode 是如何工作的。通过示例读者可以发现 Chaincode 是通过 peer 模块的 Chaincode 子命令来完成相关部署的工作的, 本节将详细介绍这些命令的使用方法以及作用。执行命令 `peer chaincode --help` 可以获取 Chaincode 子命令的下级命令和相关参数。

`peer chaincode --help` 的执行结果如下:

```
Available Commands:
    install
    instantiate
    invoke
    list
    package
    query
    signpackage
    upgrade

Flags:
    --cafile
    -o, --orderer
    --tls
    --transient

Global Flags:
    --logging-level
    --test.coverprofile
    -v, --version
```

### 1. Chaincode 命令的公用选项

peer 模块的 Chaincode 命令一共有四个公用选项, 这些选项所有的子命令都可以使用, 这些公用选项的作用如下:

- `--cafile`: PEM 格式证书的位置。
- `-o, --orderer`: orderer 服务器的访问地址。
- `--tls`: 使用 orderer 的 TLS 证书的位置。
- `--transient`: JSON 参数的编码映射。

## 2. Chaincode 命令的子命令及其选项

Chaincode 命令的运行需要一些参数，这些参数可以是配置文件也可以是环境变量，由于涉及的参数并不是很多，因此大多数时候都会采用环境变量的方式来设置参数。

### (1) install

install 命令负责安装 Chaincode，在这个过程中如果 Chaincode 的源代码存在语法错误，install 命令会报错。install 命令的选项如下所示：

- `-c, --ctor` JSON 格式的构造参数，默认值是 “{}”。
- `-l, --lang` 编写 Chaincode 的编程语言，默认值是 “golang”。
- `-n, --name` Chaincode 的名字。
- `-p, --path` Chaincode 源代码的路径。
- `-v, --version` 当前的操作的 Chaincode 的版本，适用这些命令：install/instantiate/upgrade。

install 命令的调用示例如下所示：

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer chaincode install -n qlkszzncc -v 1.1 -p github.com/qklszzl/chaincodestudy/
simpledemo
```

install 命令成功之后，会在 peer 模块的数据文件中生成一个有 `-n` 参数和 `-v` 参数组成的文件，在本例中该文件的路径如下：

```
/opt/hyperledger/peer/production/chaincodes/
```

在该路径下面会发现一个名为 `qlkszzncc.1.1` 的文件，这个文件就是 Chaincode 打包之后的文件。

### (2) instantiate

instantiate 可以对已经执行过 `install` 命令的 Chaincode 进行实例化，instantiate 命令执行完成之后会启动 Chaincode 运行的 Docker 镜像，同时 instantiate 命令还会对 Chaincode 进行初始化。instantiate 命令的选项如下所示：

- `-C, --channelID`: 当前命令运行的通道，默认值是 “testchainid”。
- `-c, --ctor`: JSON 格式的构造参数，默认值是 “{}”。

- -E, --escv: 应用于当前 Chaincode 的系统背书 Chaincode 的名字。
- -l, --lang: 编写 Chaincode 的编程语言, 默认值是 “golang”。
- -n, --name: Chaincode 的名字。
- -P, --policy: 当前 Chaincode 的背书策略。
- -v, --version: 当前操作的 Chaincode 的版本, 适用于 install/instantiate/upgrade 等命令。
- -V, --vscc: 当前 Chaincode 调用的验证系统 Chaincode 的名字。

instantiate 命令的调用示例如下所示:

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer chaincode instantiate -o orderer.qklszzn.com:7050 -C qklszzlchannel -n
qklszzncc -v 1.1 -c '{"Args":["init","a","100","b","200"]}' -P "OR
('Org1MSP.member','Org2MSP.member')"
```

instantiate 命令成功执行之后, 可以通过 docker ps 命令查看已经启动的运行 Chaincode 的 docker 容器。

### (3) invoke

invoke 命令用来调用 chaincode。invoke 命令的选项如下所示:

- -C, --channelID: 当前命令运行的通道, 默认值是 “testchainid”。
- -c, --ctor: JSON 格式的构造参数, 默认值是 “{}”。
- -n, --name: Chaincode 的名字。

invoke 命令的调用方式如下所示:

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer chaincode invoke -o 192.168.23.212:7050 -C qklszzlchannel -n qklszzncc -c
'{"Args":["invoke","set","akeym","11234343"]}'
```

### (4) list

list 命令用来查询已经安装的 Chaincode, list 命令的选项如下所示:

- -C, --channelID: 当前命令运行的通道, 默认值是 “testchainid”。
- --installed: 获取当前 Peer 节点已经被 install 的 chaincode。
- --instantiated: 获取当前 Channel 中已经被 instantiated 的 chaincode。

调用示例:

```
peer chaincode list --installed
```

### (5) package

package 用来将 Chaincode 打包。package 命令的选项如下所示：

- -s, --cc-package: 对打包后的 Chaincode 进行签名。
- -c, --ctor: JSON 格式的构造参数，默认值是 “{}”。
- -i, --instantiate-policy: Chaincode 的权限。
- -l, --lang: 编写 Chaincode 的编程语言，默认值是 “golang”。
- -n, --name: Chaincode 的名字。
- -p, --path: Chaincode 源代码的路径。
- -S, --sign: 对打包的文件用本地的 MSP (core.yaml 配置文件中的 localMspId 指定的值) 进行签名。
- -v, --version: 当前操作的 Chaincode 的版本，适用于 install/instantiate/upgrade 等命令。

package 命令的调用方式如下所示：

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp

peer chaincode package -p github.com/hyperledger/fabric/examples/chaincode/go/
chaincode_example02 -n r_test_cc6 -v 1.1 -s -S -i "OR ('Org1MSP.member', 'Org2MSP.
member')" r_test_cc6_1.1.out
```

### (6) query

query 命令用来执行 Chaincode 代码中的 query 方法。query 命令的选项如下所示：

- -C, --channelID: 当前命令运行的通道，默认值是 “testchainid”。
- -c, --ctor: JSON 格式的构造参数，默认值是 “{}”。
- -x, --hex: 是否对输出的内容进行编码处理。
- -n, --name: Chaincode 的名字。
- -r, --raw: 是否输二进制内容。
- -t, --tid: 指定当前查询的编号。

query 命令调用示例如下所示：

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp

peer chaincode query -C qklszzlchannel -n r_test_cc6 -c '{"Args":["query","a"]}'
```



### (7) signpackage

signpackage 用来对已经打好包的 Chaincode 进行签名。

调用示例:

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer chaincode signpackage r_test_cc6_1.1.out sign_r_test_cc6_1.1.out
```

### (8) upgrade

upgrade 命令用来更新已经存在的 Chaincode。upgrade 命令的选项如下:

- -C, --channelID: 当前命令运行的通道, 默认值是“testchainid”。
- -c, --ctor: JSON 格式的构造参数, 默认值是“{}”。
- -E, --escv: 应用于当前 Chaincode 的系统背书 Chaincode 的名字。
- -l, --lang: 编写 Chaincode 的编程语言, 默认值是“golang”。
- -n, --name: Chaincode 的名字。
- -p, --path: Chaincode 源代码的路径。
- -P, --policy: 当前 Chaincode 的背书策略。
- -v, --version: 当前的操作的 Chaincode 的版本, 适用于 install/instantiate/upgrade 等命令。
- -V, --vscc: 当前 Chaincode 调用的验证系统 Chaincode 的名字。

upgrade 命令的调用示例如下所示:

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer chaincode upgrade -o orderer.example.com:7050 -n r_test_cc6 -v 1.1 -C
qklszzlchannel -c '{"Args":["init","a","100","b","200"]}'
```

## 7.5 如何通过 Chaincode 进行交易的 endorse

在第 1 章里面我们介绍了区块链的基本技术特点, 其中一个重要的特点是: 区块链是一个去中心的, 所有参与方集体维护的公共账本。Fabric 作为一个典型的区块链的技术平台当然也具备这样的特点。Fabric 中对数据参与方对数据的确认真实通过 Chaincode 来进行的。

在 Fabric 中有一个非常重要的概念称为 Endorsement, 中文名为背书。背书的过程是一笔交易被确认的过程。而背书策略被用来指示对相关的参与方如何对交易进行确认。当一个节点接收到一个交易请求的时候, 会调用 VSCC (系统 Chaincode, 专门负责处理背书相关

的操作)与交易的 Chaincode 共同来验证交易的合法性。在 VSCC 和交易的 Chaincode 共同对交易的确认中,通常会做以下的校验。

- 所有的背书是否有效(参与的背书的签名是否有效)。
- 参与背书的数量是否满足要求。
- 所有背书参与方是否满足要求。

背书政策是指定第二和第三点的一种方式。这些概念看起来还是比较难懂的,理解它们最好的办法是通过一个具体的实例。背书策略的设置是通过 Chaincode 部署时 instantiate 命令中 -P 参数来设置的。命令样式如下:

```
peer chaincode instantiate -o orderer.qklszzn.com:7050 -C qklszzlchannel12 -n
cc_endfinlshed -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "AND ('Org1MSP.
member','Org2MSP.member')"
```

上述命令是对 Chaincode 进行实例化的操作,我们提取 -P 后面的参数:

```
AND ('Org1MSP.member','Org2MSP.member')
```

这个参数包说明的是当前 Chaincode 发起的交易,需要组织编号为 Org1MSP 和组织编号为 Org2MSP 的组织中的任何一个用户共同参与交易的确认并且同意,这样交易才能生效并被记录到区块链中。通过上述背书策略的实例我们可以知道背书策略是通过一定的关键字和系统的属性组成的。根据 Fabric 的系统定义,可以将上面的背书策略拆解如下:

- AND 参与背书者之间的关系,AND 表示所有参与方共同对交易进行确认。除了 AND 之外还可以使用关键字 OR,如果使用关键字 OR 表示参与方的任何一方参与背书即可完成交易的确认。

Org1MSP.member 这是表示参与背书的组织和组织中参与背书的用户。Org1MSP 表示组织的编号,这个值是怎么来的呢?在 5.2.1 节中我们介绍过 cryptoge 模块,该模块根据配置文件生成系统的配置和账号信息。在 cryptogen 的配置文件中有一个节点 Organizations->ID,该节点的值就是该组织的编号,也是在配置背书策略时需要用到的组织的编号。member 泛指组织内的任何一个用户,当然也可以是组织某个具体的用户。

通过上面的描述,基本上可以了解背书策略的编写规则,下面通过几个实例进一步了解背书策略的编写规则。

#### 背书规则示例一

```
AND('Org1MSP.member','Org2MSP.member','Org3MSP.member')
```

按照背书规则示例一进行背书的交易,必须经过组织 Org1MSP、Org2MSP、Org3MSP 中的用户共同验证交易才能生效。

#### 背书规则示例二

```
OR('Org1MSP.member','Org2MSP.member')
```

按照背书规则示例二进行背书的交易,只需要经过组织 Org1MSP 或者 Org2MSP 中的任何一个成员验证,即可生效。

### 背书规则示例三

```
OR('Org1MSP.member', AND('Org2MSP.member', 'Org3MSP.member'))
```

按照背书规则示例三进行背书的交易有两种办法让交易生效:

- 组织 Org1MSP 中的某个用户对交易进行验证。
- 组织 Org2MSP 和 Org3MSP 中的成员共同对交易进行验证。

以上介绍了背书规则。有一点需要注意的,背书规则只针对 Chaincode 中写入数据的操作进行校验,对于查询类的操作不需要背书。以 Golang 版本的 chaincode 为例,需要利用背书规则对操作进行校验的方法如下:

```
PutState(key string, value []byte) error  
DelState(key string) error
```

Fabric 中的背书是发生在客户端的,需要进行相关的代码的编写才能完成整个背书的操作。在本书的第 8 章会专门介绍 Fabric 客户端 SDK,在 Node.js、Golang、Java 等语言 SDK 介绍中都会有专门的小节介绍背书相关的内容。

## 7.6 Chaincode 的调试方法

在前面的章节中介绍了 Chaincode 源代码的编写和部署,通过这些内容可以发现,Chaincode 是运行在 Docker 容器中的,这样的好处是 Chaincode 的运行环境比较安全,受到外界的干扰相对比较小。但是这样有个坏处是如果需要查看 Chaincode 的运行日志是需要进入 Docker 容器中,而且这样的方式也不利于程序的调试。那么 Chaincode 在 Docker 容器外面是否可以运行呢? Chaincode 可以像 Java 等语言一样在相关的 IDE 中进行 Debug 吗?这两个问题的答案都是肯定的。下面详细介绍这两个问题的解决方案。

### 7.6.1 Chaincode 在 Docker 容器之外的运行

Chaincode 是通过 Golang 编写的一段代码,而且根据 Chaincode 的代码规范,一个 Chaincode 代码必须包含一个 main 函数,因此任何 Chaincode 代码都可以编译一个可执行文件,然后单独运行,通过以下步骤可以实现这一目标。

#### 第一步:注册需要调试的 chaincode

如果需要在 Docker 容器之外运行 Chaincode,首先需要把需要运行的 Chaincode 注册到 Fabric 中,注册完成后就可以借用已注册成功的 chaincode 的名字和版本号在 Docker 容器之



外运行 Chaincode。这里有一点需要强调，在注册的时候只需要向 Fabric 提交 Chaincode 的名字和版本号，实际注册的 Chaincode 的代码并不重要。注册步骤实际上是执行 Chaincode 的 install 和 instantiate 操作。注册步骤如下：

### （1）设置 Peer 节点的运行模式

可以通过修改配置文件或者添加环境变量的方式修改 Peer 节点的启动模式。配置文件可以通过修改 core.yaml 文中的 chaincode 节点的 mode 子节点的值，将 mode 的值修改为 net，修改后如下所示：

```
chaincode:
  mode: net
```

或者通过环境变量修改 Peer 运行模式：

```
export set CORE_CHAINCODE_MODE=net
```

### （2）注册 Chaincode

```
export set FABRIC_CFG_PATH=/opt/hyperledger/peer
export set ORDERER_GENERAL_LOGLEVEL=debug
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

```
peer chaincode install -n qklszzncc -v 1.1 -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02
```

```
peer chaincode instantiate -o orderer.qklszzn.com:7050 -C qklszzlchannel -n
qklszzncc -v 1.0 -c '{"Args":["init","a","100","b","200"]}'
```

注册成功之后一定要记住 install 命令执行时参数 -n 和 -v 的值，这一点非常重要，后面的配置中需要用到这两个参数的值。

### 第二步：将 Peer 节点设置为调试模式

要想在 Docker 外面运行 Chaincode，首先需要调整 Peer 节点的运行模式。可以通过修改配置文件或者添加环境变量的方式修改 Peer 节点的启动模式。配置文件可以通过修改配置文件 core.yaml 中的 chaincode 节点的 mode 子节点的值，将 mode 的值修改为 dev，修改后如下所示：

```
chaincode:
  mode: dev
```

mode 子节点有两个属性 dev 和 net，net 表示 chaincode 运行在 Docker 容器中，dve 表示 Chaincode 运行在容器之外，通常用于开发模式。

除了配置文件还可以通过在 Peer 模块启动的时候设置环境变量的方式让 Chaincode 可



以运行在 Docker 容器之外，环境变量如下：

```
export set CORE_CHAINCODE_MODE=dev
```

如果设置的 dev 模式，那么当前 Peer 模块不能执行 peer chaincode instantiate 命令

### 第三步：编译 chaincode

我们以 8.2 节中的代码为例：

```
$GOPATH/src/qklszzl/chaincodestudy/simpledemo
go build simplechaincode.go

// MacOS 系统请执行如下代码
go build -ldflags -s simplechaincode.go
```

### 第四步：运行 Chaincode

现在可以运行 Chaincode，要运行 Chaincode 需要设置相应的环境变量和系统参数，运行的命令如下：

```
export set CORE_PEER_ADDRESS=192.168.23.212:7051
export set CORE_CHAINCODE_ID_NAME=qklszzncc:1.1
export set CORE_CHAINCODE_LOGGING_LEVEL=debug
export set CORE_CHAINCODE_LOGGING_SHIM=debug

./simplechaincode -peer.address=192.168.23.212:7052
```

上述方法的环境变量和命令选项作用如下：

- CORE\_PEER\_ADDRESS：Peer 节点的 IP 地址。
- CORE\_CHAINCODE\_ID\_NAME：chaincode 的名字和版本号，下划线前面的是 chaincode 的名字，后面是 chaincode 的版本。这两个值必须和第一步 install 命令中 -n 和 -v 的参数完全一致，否则 Chaincode 无法执行。
- CORE\_CHAINCODE\_LOGGING\_LEVEL：chaincode 系统的日志级别。
- CORE\_CHAINCODE\_LOGGING\_SHIM：shim 的日志的级别。
- -peer.address=：Peer 节点中的 chaincode 的监听接口，该接口在 core.yaml 文件中的 peer 节点下面的 chaincodeListenAddress 子节点。

环境变量 CORE\_CHAINCODE\_ID\_NAME 的值非常重要，一定要和第一步 install 命令中 -n 和 -v 的参数，这一点非常重要也是非常容易引起错误的地方。

### 第五步：调用 Chaincode

上面启动的 Chaincode 通过下面的命令调用：

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabricconfig/crypto-
config/peerOrganizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

```
peer chaincode invoke -o 192.168.23.212:7050 -C qklszzlchannel -n
sampledemo5_9 -c '{"Args":["invoke","set","akeym","11234343"]}'
```

其中 -n 的值和第三步中环境变量 CORE\_CHAINCODE\_ID\_NAME 的值的下划线 ( \_ ) 前面的部分是相同的。

## 7.6.2 Chaincode 在 IDE 中的调试

上一节介绍了如何在 Docker 容器之外运行 Chaincode，但是如果需要开发的 Chaincode 比较复杂，可能需要借助以下 IDE 工具进行 Debug，这样更能提高开效率。本节作者以 Golang EPA 集成开发环境为例演示如何调试 Chaincode。

第一步：打开 Chaincode 的源代码。

打开 golang EPA，单击左上角的 Open 按钮（或者通过菜单 File → Open），打开 Chaincode 源代码所在的文件夹。如图 7-1 所示。

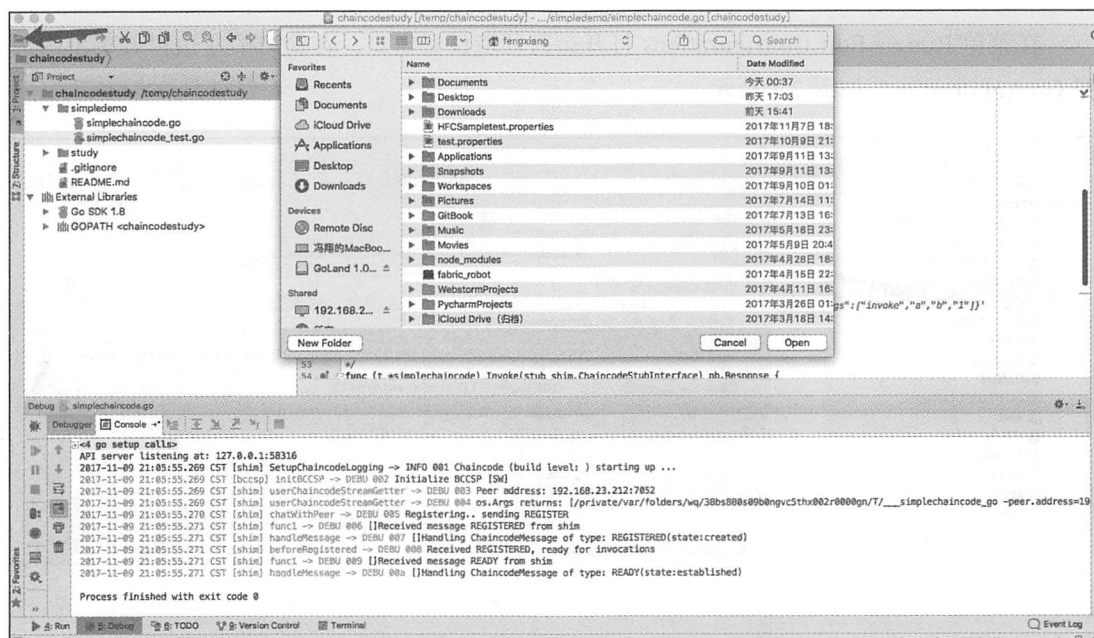


图 7-1 Golang 源码路径选择示意图

第二步：打开 Chaincode 源代码设置调试环境。

在左边的导航栏中，找到 Chaincode 源码并双击打开，如图 7-2 所示。

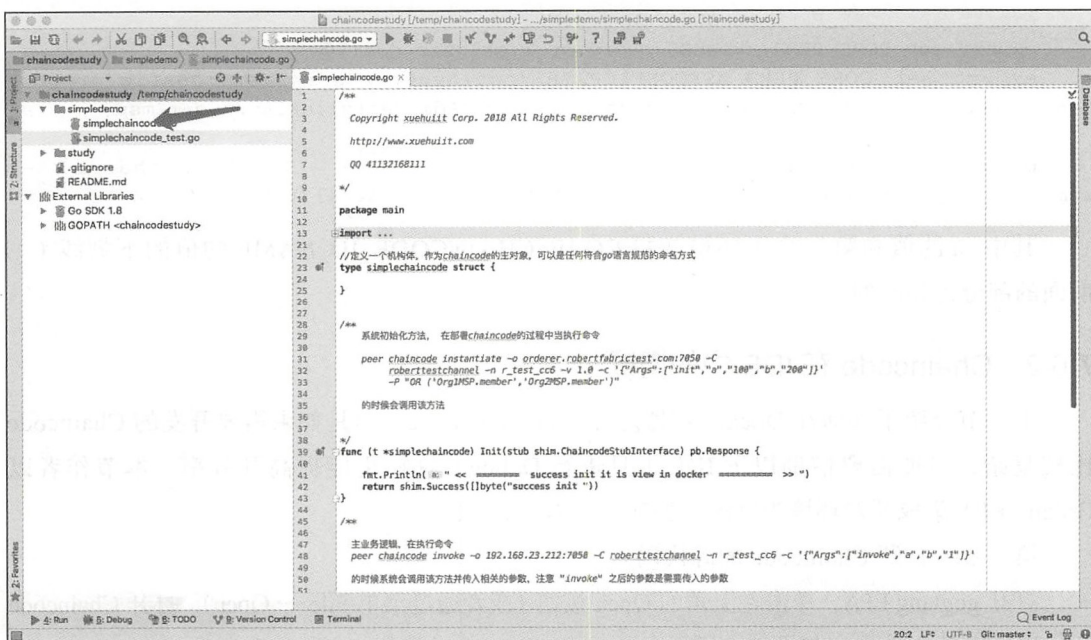


图 7-2 Golang 源码编辑示意图

在菜单 Run → Edit Configurations 打开运行环境设置窗口，在窗口的左边树形菜单中选择 Go Application，如图 7-3 所示。

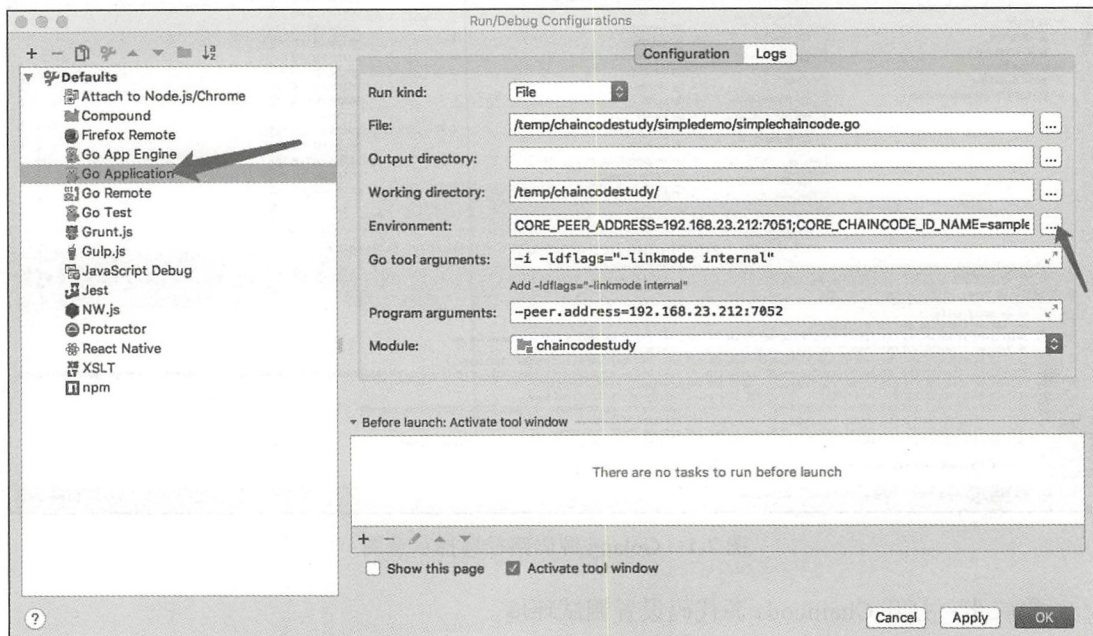


图 7-3 Golang 调试环境设置示意图



然后设置如下参数：

- File：选择需要编译的 Chaincode 的路径。
- Environment：该选择需要设置环境变量，单击右边的小按钮，然后输入下面的环境变量。

```
CORE_PEER_ADDRESS=192.168.23.212:7051
CORE_CHAINCODE_ID_NAME=sampldemo5_9:1.1
CORE_CHAINCODE_LOGGING_LEVEL=debug
CORE_CHAINCODE_LOGGING_SHIM=debug
```



注意 环境变量中不要有空，否则可能无效。

- go tool arguments：单击下面的“Add -ldflags="-linkmode internal"”。
- Progame arguments：-peer.address=192.168.23.212:7052。



注意 上述所有参数中的 IP 地址请根据自己的实际环境设置。

通过上述设置，可以直接在 golang EPA 中运行 Chaincode。

第三步：debug。

如果需要 Debug，可以在相关代码处设置断点，然后单击上面的调试按钮就可以调试了。如图 7-4 所示。

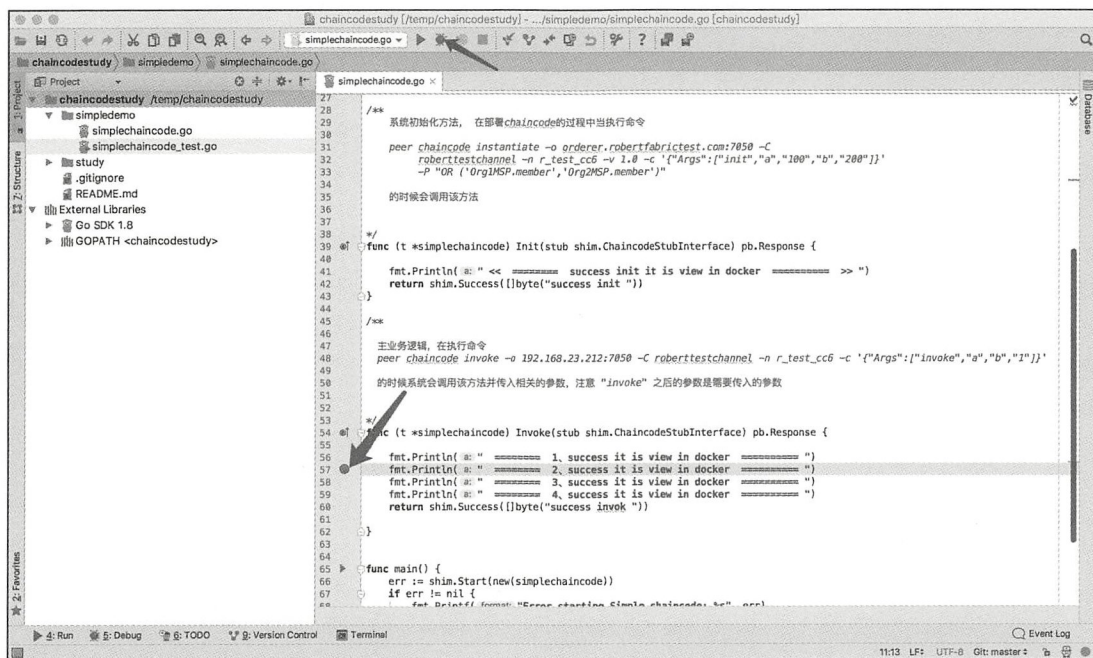


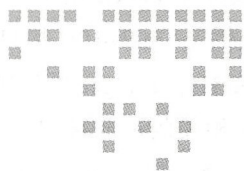
图 7-4 Golang 代码调试示意图



在编写业务逻辑比较复杂的 Chaincode 的时候，借助 IDE 工具的帮助能够起到事半功倍的效果。

## 7.7 本章小结

本章主要介绍了 Chaincode 的概念，代码结构以及编译、部署和调试的方法。Chaincode 是 Fabric 的核心技术之一，熟练地使用 Chaincode 相关的操作有助于开发出稳定可靠的 Fabric 系统。



## Fabric 和 Fabric-ca 的编程接口

Fabric 的 Peer 节点和 Orderer 节点都提供了基于 Grpc 协议的接口，通过这些接口可以和 Peer 节点以及 Orderer 节点进行交互。为了简化开发 Hyperledger 项目组实现了部分语言的 Fabric 客户端 SDK，通过这些 SDK 可以非常方便地访问 Peer 节点和 Orderer 节点。本章重点介绍这些 SDK 的使用方式。

### 8.1 Fabric 接口的通信协议和功能划分

#### 1. Grpc 协议简介

Grpc 一开始由 Google 开发，是一款语言中立、平台中立、开源的远程过程调用 RPC 系统。目前 Grpc 提供 Java、Go、C、C++、Node.js、Python、Ruby、Objective-C、PHP 和 C# 等语言的支持。Grpc 基于 HTTP/2 标准设计，带来诸如双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特性。这些特性使得其表现出很好的性能，而且报文占用的空间也比较小。Grpc 默认使用 protocol buffers 作为接口定义语言来描述服务接口和消息结构。本书中所有的案例中一律使用 protocol buffers 作为接口定义语言。

#### 2. Fabric 模块的 Grpc 接口

在前面的章节中我们知道 Fabric 一共有 5 个模块，其中 Peer 模块和 Orderer 模块提供了 Grpc 相关的接口，但是 Orderer 模块的 Grpc 接口是给 Peer 模块等调用的，在项目开发中并不需要直接调用 Orderer 模块的 Grpc 接口，本章将不讨论 Orderer 相关的 Grpc 接

口。本章将主要介绍 Fabric 的 Peer 模块的 Grpc 接口的调用方式。除此之外由于 Fabric-ca-server 项目也提供了 JSONRPC 接口供第三方程序访问, 鉴于 Fabric-ca-server 是 Fabric 重要的组成部分, 在 Hyperledger 项目组中已经提供的 4 个语言版本的 SDK 中都完整地集成了 Fabric-ca-server 相关的访问接口。在实际的项目中 Fabric-ca-server 是 Fabric 不可或缺的组成部分, 为了让读者完整地了解 Fabric 项目中第三方应用程序是如何同 Fabric 系统进行交互的, 我们在介绍 Fabric 的 Grpc 接口调用方式的同时也会介绍 Fabric-ca-server 相关接口的调用方式。Fabric 所有的 protocol buffers 接口定义都存放的路径如下所示:

<https://github.com/hyperledger/fabric/tree/release/protos>

### 3. Fabric 的 Peer 模块的接口功能划分

Fabric 的 Peer 模块提供的 Grpc 接口按照功能大致可以分为两类: 系统管理和 Chaincode 相关操作。

#### (1) 系统管理

Peer 模块的 Grpc 接口提供了如下与系统管理相关的功能:

- 获取当前 Peer 加入了哪些 Channel
- 获取当前 Peer 加入的某个 Channel 的区块数
- 获取当前的 Peer 加入的某个 Channel 中的某个区块号, 获取区块的详细信息
- 根据当前的 Peer 加入的某个 Channel 中的某个区块的哈希值, 获取区块的详细信息
- 根据当前的 Peer 加入的某个 Channel 中交易的哈希值, 获取交易的详细信息
- 获取当前 Peer 服务器中状态为 install 的 Chaincode 的信息
- 根据当前的 Peer 加入的某个 Channel 中状态为 Instantiate 的 Chaincode 的详细信息

#### (2) Chaincode 相关操作

Peer 模块的 Grpc 接口提供了如下与 Chaincode 操作相关的功能:

- 调用 Chaincode 的 query 方法
- 调用一个已经部署的 Chaincode 的 invoke 方法

### 4. Fabric-ca-server 的 RESTAPI 接口

Fabric-ca-server 的接口方法相对简单, 常用的接口如下所示:

- 注册一个 Fabric 账号
- 加载一个已经注册过的 Fabric 账号

从原理来看, 通过 protocol buffers 的接口定义文件, 任何语言都是可以通过 Grpc 协议调用 Fabric 提供的 Grpc 接口, 但是如果一切从底层开始实现的话, 这样的开发效率是非常低的。目前 Hyperledger 项目组已提供的基于 Nodejs、Java、Go、Python 这四种语言的 SDK, 这些 SDK 对 Fabric 的常用操作进行了封装, 然后把这些操作抽象成接口提供调用服

务，这样我们直接调用这些接口就可以完成对 Fabric 的相关操作。通过使用这些已经封装好的 SDK 可以大大简化了开发工作。我们推荐读者在开发过程中直接使用这些 SDK，没有必要重复发明轮子。

## 8.2 Fabric Nodejs SDK 的使用

Fabric Nodejs SDK 是目前已知的 4 种 SDK 中相对比较完善且成熟的 SDK，很多应用都是基于 Node.js 版本的 SDK 开发的。本节我们将重点介绍利用 Node.js 快速开发基于 Fabric 的应用。

### 8.2.1 如何获取 Fabric Nodejs SDK 源代码

Nodejs 版本的 SDK 的项目路径如下：

<https://github.com/hyperledger/fabric-sdk-node>

这是该项目的源代码的位置，这份源代码中包含了所有的源代码和相关的测试用例。但是这份源代码是最新的版本，并且由于依赖很多其他的 Node.js 库，因此在项目中直接使用会比较困难。目前推荐的使用方式是通过 npm 的方式来获取稳定版本的 SDK 源代码。

目前 Fabric Nodejs SDK 还不能在 Windows 上面运行，只能在 CentOS、Ubuntu、MacOs 等 Linux 或者 Unix 类的系统中运行。目前只支持 Node.js 6.9.x 或者 8.0 以上的版本，特别是 7.x 的版本会有很多问题，读者在配置的时候一定要注意 Nodejs 的版本。



- 在 Windows 系统中可以通过虚拟机安装 Ubuntu 桌面版本的来运行
- npm 是 Node.js 的包管理工具

### 8.2.2 快速构建基于 Nodejs 的 Fabric 客户端

#### 1. 项目初始化

创建一个目录，本例的目录为 /project/ws\_node/fabric\_sdk\_node\_study。目录创建完成后，在目录创建 npm 的配置文件，配置的风格如下：

```
{
  "name": "fabric-sdk-node-study",
  "version": "0.0.1",
  "description": "fabric-sdk-node-study",
  "main": "main.js",
  "keywords": [
```



```

    "v1.0 fabric nodesdk sample",
    "alpha based nodesdk sample app"
  ],
  "engines": {
    "node": ">=6.9.5",
    "npm": ">=3.10.10"
  },
  "dependencies": {
    "fabric-ca-client": "^1.0.0",
    "fabric-client": "^1.0.0",
  },
  "license": "Apache-2.0"
}

```

然后在项目目录中执行如下命令：

```
npm install
```

执行完成之后，项目目录下面会生成一个名为 `node_modules` 的目录，该目录中存放了 Fabric-sdk-node 依赖的相关第三方包。

## 2. 一个 Node.js 访问 Fabric 的例子

和前面一样，我们还是先提供一个完整的例子，让读者能够快速了解通过 Node.js SDK 访问 Fabric 的方法，在后面会详细介绍每个步骤和相关的方法，最后会提供一个完整的例子供读者参考。一个简单实例的代码如下所示：

```

// 导入相关的包
var co = require('co');
var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
var FabricCAService = require('fabric-ca-client');

// 定义证书文件的缓存目录
var tempdir = "/project/ws_nodejs/fabric_sdk_node_studynew/fabric-client-kvs";

// fabric client 代理
var client = new hfc();

var cryptoSuite = hfc.newCryptoSuite()
cryptoSuite.setCryptoKeyStore( hfc.newCryptoKeyStore({ path:tempdir } ) )
client.setCryptoSuite(cryptoSuite)

// 创建 CA 客户端

```

```

var caClient = new FabricCAService('http://192.168.23.212:7054', null, '', cryptoSuite);

// 创建通道的客户端代理, 被创建的通道名字为: roberttestchannel12
var channel = client.newChannel('roberttestchannel12');

// 创建 order, 该 Orderer 运行的服务器地址为: 192.168.23.212, 创建成功后将该 Orderer 加入到 channel 中。
var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);

// 创建 Peer 节点的客户端代理, 该 Peer 节点的 IP 地址为 192.168.23.212 端口号为: 7051
var peer = client.newPeer('grpc://192.168.23.212:7051');
channel.addPeer(peer);

```

上面是一个很简单的例子, 运行这段代码可以获取某个 Peer 服务器中已经加入的某个 Channel 的区块的高度。下面我们将详细解释上面的代码。

不论什么语言版本的 SDK 在调用之前首先需要知道三个基本的信息: 账号、Peer 和 Orderer 服务器节点的地址。Fabric 解释 Nodejs SDK 已经封装了相关的接口类, 相关代码如下:

```

// 导入相关的包
var co = require('co');
var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
var FabricCAService = require('fabric-ca-client');

// 定义证书文件的缓存目录
var tempdir = "/project/ws_nodejs/fabric_sdk_node_studynew/fabric-client-kvs";

// fabric client 代理
var client = new hfc();

var cryptoSuite = hfc.newCryptoSuite()
cryptoSuite.setCryptoKeyStore( hfc.newCryptoKeyStore({ path:tempdir } ) )
client.setCryptoSuite(cryptoSuite)

// 创建 CA 客户端
var caClient = new FabricCAService('http://192.168.23.212:7054', null, '', cryptoSuite);

// 创建通道的客户端代理, 被创建的通道名字为: roberttestchannel12
var channel = client.newChannel('roberttestchannel12');

// 创建 order, 该 Orderer 运行的服务器地址为: 192.168.23.212, 创建成功后将该 Orderer 加入到 channel 中。

```

```

var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);
// 创建 Peer 节点的客户端代理, 该 Peer 节点的 IP 地址为 192.168.23.212 端口号为: 7051
var peer = client.newPeer('grpc://192.168.23.212:7051');
channel.addPeer(peer);

co(( function *() {

    let member = yield getOrgUser4Local();
    // let member = yield getOrgUser4FabricCa("user88","peer2wd");
    let resultpeerinfo = yield channel.queryInfo(peer)
    console.info( JSON.stringify(resultpeerinfo) )

})

)())

/**
 *
 * 通过 CA 获取当前用户的证书信息
 *
 * @param username
 * @param password
 * @returns {Promise.<TResult>}
 */
function getOrgUser4FabricCa(username,password) {

    var member

    return hfc.newDefaultKeyValueStore({path:tempdir})
        .then( (store)=>{

            client.setStateStore(store);
            client._userContext = null;

            return client.getUserContext(username,true).then( (user)=>{

                if( user && user.isEnrolled() ){

                    console.info(` success enrolled admin `)
                    return user;

                } else{

                    return caClient.enroll( {enrollmentID: username, enrollmentSecret:
password} ).then(

                        (enrollment)=>{

```

```

        console.info('Successfully enrolled user \'' + username
+ '\');

        member = new User(username)
        member.setCryptoSuite( client.getCryptoSuite() )

        return member.setEnrollment( enrollment.key,enrollment.
certificate,'Org1MSP' )

    }).then( ()=>{

        return client.setUserContext(member)

    }).then(()=>{
        return member
    })
    .catch((err)=>{
        console.error('enroll admin error'+err.stack)
        return null
    })
    })

    })

    })

}

/**
 * 根据 cryptogen 模块生成的账号通过 Fabric 接口进行相关的操作
 */
function getOrgUser4Local() {

    var keyPath = "/project/fabric_resart/config_demo/org1/186/fabric-user/msp/
keystore";
    var keyPEM = Buffer.from(readAllFiles(keyPath)[0]).toString();
    var certPath = "/project/fabric_resart/config_demo/org1/186/fabric-user/msp/
signcerts";
    var certPEM = readAllFiles(certPath)[0].toString();

    return hfc.newDefaultKeyValueStore({

        path:tempdir

    }).then((store) => {
        client.setStateStore(store);

```



```

    return client.createUser({
      username: 'user87',
      mspid: 'Org1MSP',
      cryptoContent: {
        privateKeyPEM: keyPEM,
        signedCertPEM: certPEM
      }
    });
  });
};

```

上面的代码创建了 Peer 节点、Orderer 节点、Fabrica 节点的客户端代理, 接下来就需要通过这些客户端代理进行相关的操作。在这些操作之前首先需要确定操作的账号, 在本书的第 6 章介绍过 Fabric 的账号体系, Fabric 的有两种方法获取账号, 分别是 cryptogen 模块根据配置文件生成账号和 Fabric-ca-server 服务器生成账号。上述示例中的两个方法分别介绍了这两种获取方法, 具体可以参考上述代码的注释。

有了 Peer、Orderer 以及 Fabric-ca 的客户端代理, 在获取账号之后便可以访问 Peer 服务器了。下面的示例代码中演示了如何获取 Channel 中区块数:

```

co(( function *() {

  let member = yield getOrgUser4Local();
  let resultpeerinfo = yield channel.queryInfo(peer)
  console.info( JSON.stringify(resultpeerinfo) )

})

)())

```



**注意** co 是 Nodejs 的一个异步框架, 可以有效解决 Nodejs 的回调陷阱问题。

这样我们就完成了一个最简单的利用 Nodejs 访问 Fabric 的例子。通过上面的例子我们发现通过使用 Fabric Nodejs SDK 访问 Fabric 可以大大简化开发工作。

### 3. Fabric Nodejs SDK 提供的常用方法

上面的代码中我们简单地描述了如何通过使用 Fabric Nodejs SDK 访问 Fabric, 我们将在上面代码的基础上介绍其他的一些功能。

以下实例代码都只能在 `co(( function *() { })())` 函数中运行。

#### (1) 系统管理

- 获取当前 Peer 加入了哪些 Channel

```
let resultchannels = yield client.queryChannels(peer)
console.info( JSON.stringify( resultchannels ) )
```

- 获取当前 Peer 加入的某个 Channel 的区块数

```
let resultpeerinfo = yield channel.queryInfo(peer)
console.info( JSON.stringify(resultpeerinfo) )
```

- 根据当前的 Peer 加入的某个 Channel 中的某个区块号, 获取区块的详细信息

```
let blockinfoNum = yield channel.queryBlock(23, peer, null);
console.info( JSON.stringify( blockinfoNum ) )
```

- 根据当前的 Peer 加入的某个 Channel 中的某个区块的哈希值, 获取区块的详细信息

```
let blockinfohash = yield channel.queryBlockByHash(new Buffer("hashcode", "hex"), peer)
console.info( JSON.stringify(blockinfohash) )
```

- 根据当前的 Peer 加入的某个 Channel 中交易的哈希值, 获取交易的详细信息

```
let resulttxinfo = yield channel.queryTransaction("56f51f9a54fb4755fd68c6c2493
1234a59340f7c98308374e9991d276d7d4a96", peer);
console.info( JSON.stringify( resulttxinfo ) )
```

- 获取当前 Peer 服务器中状态为 install 的 Chaincode 的信息

```
let resultchannels = yield client.queryInstalledChaincodes(peer)
console.info( JSON.stringify( resultchannels ) )
```

- 根据当前的 Peer 加入的某个 Channel 中状态为 Instantiate 的 Chaincode 的详细信息

```
let chaincodeinstalls = yield channel.queryInstantiatedChaincodes( peer )
console.info( JSON.stringify( chaincodeinstalls ) )
```

## (2) Chaincode 相关操作

- 调用 Chaincode 的 query 方法

```
tx_id = client.newTransactionID();
var request = {
  chaincodeId: "cc_endorse",
  txId: tx_id,
  fcn: "invoke",
  args: ["GetTxID", "akey", "3333333333 hahaha 1st key ddddddd this is last version"]
};
let chaincodequeryresult = yield channel.queryByChaincode( request , peer );
```

- 调用 Chaincode 的 invoke 方法

```
let tx_id = client.newTransactionID();
var request = {
  targets: peer,
  chaincodeId: "cc_endorse1",
```

```

        fcn: "invoke",
        args: ["a", "b", "1"],
        chainId: "roberttestchannel",
        txId: tx_id
    });

    let chaincodeinvokresult = yield channel.sendTransactionProposal(request);

    var proposalResponses = chaincodeinvokresult[0];
    var proposal = chaincodeinvokresult[1];
    var header = chaincodeinvokresult[2];
    var all_good = true;

    for (var i in proposalResponses) {

        let one_good = false;
        if (proposalResponses && proposalResponses[0].response &&
            proposalResponses[0].response.status === 200) {
            one_good = true;
            console.info('transaction proposal was good');
        } else {
            console.error('transaction proposal was bad');
        }
        all_good = all_good & one_good;
    }

    if (all_good) {

        console.info(util.format(

            'Successfully sent Proposal and received ProposalResponse: Status
- %s, message - "%s", metadata - "%s", endorsement signature: %s',
            proposalResponses[0].response.status, proposalResponses[0].response.
message,

            proposalResponses[0].response.payload, proposalResponses[0].endorsement
            .signature));

        var request = {
            proposalResponses: proposalResponses,
            proposal: proposal,
            header: header
        };

        var transactionID = tx_id.getTransactionID();
        var sendPromise = yield channel.sendTransaction(request);

    }

```

```
console.info(all_good)
```

#### 4. 如何通过 Fabric Nodejs SDK 完成背书交易

在 Fabric 共享一个账本，为了保证数据不被篡改，有时候一笔交易需要两个不同的组织同时确认才能让这笔交易生效。这个过程称为背书（endorsement），背书的过程是在客户端完成的，由于背书是 Fabric 非常重要的一个概念，我们将背书的过程作为一个部分单独来说明。

在 Fabric 中如果一个交易需要两个或两个以上的组织同时确认才能生效，那么在部署 Chaincode 的时候需要说明哪些组织需要参与到交易的确认中。部署两个组织同时参与确认的命令示例如下：

```
peer chaincode instantiate -o orderer.robertfabrictest.com:7050 -C
roberttestchannel12 -n cc_endfinlshed -v 1.0 -c '{"Args":["init","a","100",
"b","200"]}' -P "AND ('Org1MSP.member','Org2MSP.member')"
```

注意上面的命令中，以前示例经常用到的 OR 换成了 AND，这说明如果调用这个 Chaincode 完成一笔交易，需要 Org1MSP 和 Org2MSP 这两个的组织的任意账号同时完成确认才有可能完成交易。通过 Fabric Nodejs SDK 可以非常容易的完成背书的过程，只需要在 Chaincode 的 Invoke 方法之前将需要参与背书的 Peer 节点添加到发起交易的 Channel 中即可。代码如下：

```
var peer = client.newPeer('grpc://192.168.23.212:7051');
channel.addPeer(peer);
```

就是这么简单，这也是使用 Nodejs SDK 的好处，如果从头开始那就比较麻烦了。如果有多个需要参与交易确认的组织，将这些组织提供的背书节点加入到当前的通道中就行了



**注意** 参与背书的 Peer 节点必须安装相同版本的 Chaincode，否则无法完成背书。

### 8.2.3 Fabric Nodejs SDK 中 TLS 的设置

如果 Orderer 节点和 Peer 节点都启用了 TLS 模式，那么客户端也需要进行相关的设置，Nodejs 中设置 TLS 的代码样例如下所示：

#### 1. Orderer 节点的设置

```
// 设置证书文件
let tls_cacerts_content = fs.readFileSync('/project/opt_fabric/
fabricconfig/crypto-config/ordererOrganizations/robertfabrictest.com/orderers/
orderer.robertfabrictest.com/tls/ca.crt');
```



```
let opt = {
    pem: Buffer.from(tls_cacerts_content).toString(),
    'ssl-target-name-override': 'orderer.robertfabrictest.com'
}

var peer = client.newPeer( 'grpc://192.168.23.212:7051' , opt );
channel.addPeer( peer );
```

## 2. Peer 节点的设置

```
let tls_cacerts_content = fs.readFileSync('/project/opt_fabric/
fabricconfig/crypto-config/peerOrganizations/org1.robertfabrictest.com/peers/
peer0.org1.robertfabrictest.com/tls/ca.crt');
```

```
let opt = {
    pem: Buffer.from(tls_cacerts_content).toString(),
    'ssl-target-name-override': 'peer0.org1.robertfabrictest.com'
}

var order = client.newOrderer( 'grpc://192.168.23.212:7050', opt );
channel.addOrderer( order );
```

设置 TLS 之后, Orderer 节点和 Peer 节点中的访问路径中应该将 grpc 替换成 grpcs。

## 8.3 Fabric Java SDK

Fabric Java SDK 的接口和 Nodejs 比较相似, 是基于同一份 Protocol Buffers 接口定义语言进行开发的。但是由于编程语言之间的差异, 具体的接口名称会存在一些不一样的地方。关于接口的说明这里不再重复, 具体可以参考 Fabric Nodejs 部分的说明, 这里直接介绍如何使用。

### 8.3.1 Fabric Java SDK 的安装

Fabric Java SDK 使用之前需要安装, 下面是 Fabric Java SDK 的安装步骤。

#### 第一步: 环境准备

在编译 Fabric 之前请先安装 JDK 和 Maven, JDK 版本推荐 1.8 或以上, Maven 的版本推荐 3.5 或以上。

#### 第二步: 下载并且编译

编译 Fabric Java SDK 有两种方法。如果需要在已有的基于的 Maven 的项目中集成 Fabric Java SDK, 可以在 Maven 的 pom.xml 文件中增加以下内容:

```
<dependencies>
    <dependency>
```

```

        <groupId>org.hyperledger.fabric-sdk-java</groupId>
        <artifactId>fabric-sdk-java</artifactId>
        <version>1.1.0-SNAPSHOT</version>
    </dependency>
</dependencies>

```

同时在 Maven 的 Setting.xml 文件中增加以下内容

```

<profiles>
  <profile>
    <id>allow-snapshots</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <repositories>
      <repository>
        <id>snapshots-repo</id>
        <url>https://oss.sonatype.org/content/repositories/snapshots</url>
        <releases>
          <enabled>false</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </repository>
    </repositories>
  </profile>
</profiles>

```

通过添加上述内容后执行 Maven 相关的系统会自动下载相关的 jar 包。

如果项目不是基于 Maven 构建的，那么可以下载 Fabric Java SDK 的源代码编译，编译过程如下：

```

git clone https://github.com/hyperledger/fabric-sdk-java.git
cd fabric-sdk-java
mvn install
mvn dependency:copy-dependencies

```

上述命令执行完成后，在当前源代码的路径下面会生成一个名为 target 的文件夹，在 target 文件夹中有一个名为 fabric-sdk-\*.jar 的文件，该 jar 包就是 Fabric Java SDK 的 jar 包。同时在 target 文件夹中还会存在一个名为 dependency 的文件夹，该文件夹中包含了 Fabric Java SDK 依赖的相关 jar 包，这些 jar 文件和刚才的 fabric-sdk-\*.jar 文件一起复制到项目的 classpath 路径下面，就可以正常调用 Fabric Java SDK 的相关接口。

### 8.3.2 Fabric Java SDK 的常用接口

使用 Fabric Java SDK 访问 Fabric 是非常简单的，本节将详细介绍 Fabric Java SDK 的技

术特性和使用方法。

### 1. 一个简单的 Java 访问 Fabric 的例子

我们首先还是通过一个简单完整的例子, 让读者对 Fabric Java SDK 有一个直观的了解。

示例代码如下所示:

```
package com.onechain.fabric.test;
import org.apache.commons.io.IOUtils;
import org.bouncycastle.asn1.pkcs.PrivateKeyInfo;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.openssl.PEMParser;
import org.bouncycastle.openssl.jcajce.JcaPEMKeyConverter;
import org.hyperledger.fabric.sdk.*;
import org.hyperledger.fabric.sdk.exception.*;
import org.hyperledger.fabric.sdk.security.CryptoSuite;
import org.hyperledger.fabric_ca.sdk.HFCAClient;
import org.hyperledger.fabric_ca.sdk.exception.EnrollmentException;
import java.io.*;
import java.lang.reflect.InvocationTargetException;
import java.net.MalformedURLException;
import java.security.*;
import java.security.spec.InvalidKeySpecException;
import java.util.Set;
import static java.lang.String.format;

/**
 * @author robert.feng
 * <p>
 * QQ: 411321681
 */
public class FabricJavaTest {
    // 设置加密方式
    static {
        Security.addProvider(new org.
            bouncycastle.jce.provider.BouncyCastleProvider());
    }
    public static void main(String[] args) {
        try {
            // 创建客户端代理
            HFClient hfclient = HFClient.createNewInstance();
            CryptoSuite cryptosuite =
                CryptoSuite.Factory.getCryptoSuite();
            hfclient.setCryptoSuite(cryptosuite);

            // 设置用户
            //User user = getFabricUser4Local("user88", "org1", "Org1MSP");
            User user = getFabricUser4FabricCA("user88", "org1", "Org1MSP");
            hfclient.setUserContext(user);

            // 创建通道的客户端代理
            Channel channel = hfclient.newChannel("roberttestchannel12");
```

```

// 创建 orderer 服务器客户端代理并加到通道中
Orderer orderer = hfclient.newOrderer("orderer", "grpc://192.168.23.212:7050");
channel.addOrderer(orderer);

// 创建 Peer 服务器节点的客户端代理并加入到通道中
Peer peer = hfclient.newPeer("peer0", "grpc://192.168.23.212:7051");
channel.addPeer(peer);
// 初始化通道
channel.initialize();
// 获取 IP 地址为: 192.168.23.212 的 Peer 服务器的 roberttestchannel12 的相关
信息, 包括前面, 区块链高度等信息
BlockchainInfo blockchianinfo = channel.queryBlockchainInfo(peer);
System.out.println(blockchianinfo.getHeight() + " ");
} catch (RuntimeException e) {
    e.printStackTrace();
}
}

// 通过 Fabric CA 获取 Fabric 账号
private static User getFabricUser4FabricCA(String username, String org, String
orgId) throws IllegalAccessException, InvocationTargetException, InvalidArgumentException,
InstantiationException, NoSuchMethodException, CryptoException, ClassNotFoundException,
MalformedURLException, EnrollmentException, org.hyperledger.fabric_ca.sdk.exception.
InvalidArgumentException {

    FabricUsersImpl user = new FabricUsersImpl(username, org);
    user.setMspId(orgId);
    CryptoSuite cryptosuite = CryptoSuite.Factory.getCryptoSuite();
    HFCAClient caclient = HFCAClient.createNewInstance("http://192.168.23.212:7054",
null);
    caclient.setCryptoSuite(cryptosuite);
    Enrollment enrollment = caclient.enroll("user88", "peer2wd");
    user.setEnrollment(enrollment);
    return user;
}

// 根据 cryptogen 模块生成的账号创建 Fabric 账号
private static User getFabricUser4Local(String username, String org, String
orgId) throws IOException, NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException {
    FabricUsersImpl user = new FabricUsersImpl(username, org);
    user.setMspId(orgId);
    String certificate = new String(IOUtils.toByteArray(new FileInputStream(new
File("/project/opt_fabric/fabricconfig/crypto-config/peerOrganizations/org1.
robertfabrictest.com/users/Admin@org1.robertfabrictest.com/msp/signcerts/Admin@org1.
robertfabrictest.com-cert.pem"))), "UTF-8");
    File privatekeyfile = new File("/project/opt_fabric/fabricconfig/
crypto-config/peerOrganizations/org1.robertfabrictest.com/users/Admin@org1.
robertfabrictest.com/msp/keystore/b031338f76290f089d330b064d4534202a49ae8d65ca5d26
6c377bc46812a884_sk");
    PrivateKey privateKey = getPrivateKeyFromBytes(IOUtils.toByteArray(new

```



## 164 ❖ 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

FileInputStream(privatekeyfile)));
        EnrolleMentImpl enrollement = new EnrolleMentImpl(privateKey, certificate);
        user.setEnrollment(enrollement);
        return user;
    }
    // 配置文件中获取私钥
    private static PrivateKey getPrivateKeyFromBytes(byte[] data) throws IOException,
NoSuchProviderException, NoSuchAlgorithmException, InvalidKeySpecException {

        final Reader pemReader = new StringReader(new String(data));
        final PrivateKeyInfo pemPair;
        PEMParser pemParser = new PEMParser(pemReader);
        pemPair = (PrivateKeyInfo) pemParser.readObject();
        PrivateKey privateKey = new JcaPEMKeyConverter().setProvider(BouncyCastl-
eProvider.PROVIDER_NAME).getPrivateKey(pemPair);
        return privateKey;
    }
    // 获取私钥文件
    private static File findFileSk(String directories) {

        File directory = new File(directories);
        File[] matches = directory.listFiles((dir, name) -> name.endsWith("_sk"));
        if (null == matches) {
            throw new RuntimeException(format("Matches returned null does %s directory
exist?", directory.getAbsolutePath().getName()));
        }
        if (matches.length != 1) {
            throw new RuntimeException(format("Expected in %s only 1 sk file
but found %d", directory.getAbsolutePath().getName(), matches.length));
        }
        return matches[0];
    }

    //enrollm 实现类
    static final class EnrolleMentImpl implements Enrollment, Serializable {
        private static final long serialVersionUID = -2784835212445309006L;
        private final PrivateKey privateKey;
        private final String certificate;
        public EnrolleMentImpl(PrivateKey privateKey, String certificate) {
            this.certificate = certificate;
            this.privateKey = privateKey;
        }
        @Override
        public PrivateKey getKey() {
            return privateKey;
        }
        @Override
        public String getCert() {
            return certificate;
        }
    }
}

```

//Fabric user 实现类

```
static final class FabricUsersImpl implements User, Serializable {
    private String name;
    private Set<String> roles;
    private String account;
    private String affiliation;
    private String organization;
    private String enrollmentSecret;
    Enrollment enrollment = null;
    private String keyValStoreName;

    FabricUsersImpl(String name, String org) {
        this.name = name;
        this.organization = org;
    }
    @Override
    public String getName() {
        return this.name;
    }
    @Override
    public Set<String> getRoles() {
        return this.roles;
    }
    public void setRoles(Set<String> roles) {
        this.roles = roles;
    }
    @Override
    public String getAccount() {
        return this.account;
    }
    public void setAccount(String account) {
        this.account = account;
    }

    @Override
    public String getAffiliation() {
        return this.affiliation;
    }
    public void setAffiliation(String affiliation) {
        this.affiliation = affiliation;
    }
    @Override
    public Enrollment getEnrollment() {
        return this.enrollment;
    }
    public boolean isEnrolled() {
        return this.enrollment != null;
    }
    public String getEnrollmentSecret() {
        return enrollmentSecret;
    }
}
```

```

        public void setEnrollmentSecret(String enrollmentSecret) {
            this.enrollmentSecret = enrollmentSecret;
        }
        public void setEnrollment(Enrollment enrollment) {
            this.enrollment = enrollment;
        }
        @Override
        public String getMspId() {
            return mspId;
        }
        String mspId;
        public void setMspId(String mspID) {
            this.mspId = mspID;
        }
    }
}

```

运行上述代码可以获取指定 Channel 的区块高度。

在上述代码中一共有两个内部类, 分别是 FabricUsersImpl 和 EnrolleMentImpl。其中 FabricUsersImpl 封装了 Fabric 的账号信息, EnrolleMentImpl 是证书和密码信息的封装类, 这两个类封装了账号相关的信息。

## 2. Fabric Java SDK 常用的方法

通过上面的例子我们了解了 Fabric Java SDK 代码的结构。下面将介绍其他一些 Fabric Java SDK 的常用方法。由于篇幅的限制, 下面的示例代码我们只给出部分, 如果想要运行这些功能的代码, 可以把这些代码复制到上面示例中的 main 方法的最后。

### (1) 系统管理

- 获取当前 Peer 加入了哪些 Channel

```

Set<String> peerchannels = hfclient.queryChannels(peer);
for (String string : peerchannels) {
    System.out.println(string);
}

```

- 获取当前 Peer 加入的某个 Channel 的区块数

```

BlockchainInfo blockchianinfo = testchannel.queryBlockchainInfo(peer);
System.out.println(blockchianinfo.getHeight()+" ");

```

- 根据当前的 Peer 加入的某个 Channel 中的某个区块号, 获取区块的详细信息

```

BlockInfo blockinfo= testchannel.queryBlockByNumber(80);
ByteString blockhash = blockinfo.getBlock().getHeader().getPreviousHash();
String blockhashstr = Hex.encodeHexString(blockinfo.getPreviousHash());
System.out.println(blockhashstr);

```

- 根据当前的 Peer 加入的某个 Channel 中的某个区块的哈希值, 获取区块的详细信息

```
BlockInfo blockinfo1 = testchannel.queryBlockByHash(Hex.decodeHex("ec298dc1cd1f0e0a3f6d6e25b5796e7b5e4d668aeb6ec3a90b4aa6bb1a7f0c17").toCharArray());
System.out.println(blockinfo1.getBlock().toBuilder().toString())
```

- 根据当前的 Peer 加入的某个 Channel 中交易的哈希值，获取交易的详细信息

```
TransactionInfo transactionInfo = testchannel.queryTransactionByID("c635e06087f5f5eb632136b8e7302f34220e842950551774622b32922c25e250");
System.out.println(transactionInfo.getTransactionID());
```

- 获取当前 Peer 服务器中状态为 install 的 Chaincode 的信息

```
List<ChaincodeInfo> installchaincodes = hfclient.queryInstalledChaincodes(peer);

for (ChaincodeInfo chaincodeInfo : installchaincodes) {
    System.out.println(chaincodeInfo.getPath());
}
```

根据当前的 Peer 加入的某个 Channel 中状态为 Instantiate 的 Chaincode 的详细信息

```
List<ChaincodeInfo> instancechaincodes = testchannel.queryInstantiatedChaincodes(peer);
for (ChaincodeInfo chaincodeInfo : instancechaincodes) {
    System.out.println(chaincodeInfo.getPath());
}
```

## (2) Chaincode 相关操作

Peer 模块的 Grpc 接口提供了如下 Chaincode 相关的功能：

- 调用 Chaincode 的 query 方法

```
ChaincodeID chaincodeID = ChaincodeID.newBuilder().setName("cc_endfinlshed").build();

QueryByChaincodeRequest queryByChaincodeRequest = hfclient.newQueryProposalRequest();
queryByChaincodeRequest.setArgs(new String[]{"a"});
queryByChaincodeRequest.setFcn("query");
queryByChaincodeRequest.setChaincodeID(chaincodeID);

Collection<ProposalResponse> queryProposals;
queryProposals = testchannel.queryByChaincode(queryByChaincodeRequest);

for (ProposalResponse proposalResponse : queryProposals) {
    if (!proposalResponse.isVerified() || proposalResponse.getStatus() != Status.SUCCESS) {
        System.out.println(" Failed query proposal from peer! ");
    } else {
        String payload = proposalResponse.getProposalResponse().getResponse().getPayload().toStringUtf8();
        System.out.println(" Query success : "+payload );
    }
}
```



- 调用一个已经部署的 Chaincode 的 invoke 方法

invoker 方法相对比较复杂, 需要定一个新的方法 sendTranstion。

```
private static CompletableFuture<BlockEvent.TransactionEvent> sendTranstion(
    HFClient client,
    Channel channel,
    ChaincodeID chaincodeID,
    User user) {

    try {
        //ChaincodeID chaincodeID = ChaincodeID.newBuilder().setName
        ("sampledemo5_9").build();
        Collection<ProposalResponse> successful = new LinkedList<>();

        TransactionProposalRequest transactionProposalRequest = client.
        newTransactionProposalRequest();
        transactionProposalRequest.setChaincodeID(chaincodeID);
        transactionProposalRequest.setFcn("invoke");
        transactionProposalRequest.setArgs(new String[] {"a","b","1"});
        transactionProposalRequest.setProposalWaitTime(300000);
        transactionProposalRequest.setUserContext(user);

        Collection<ProposalResponse> invokePropResp = channel.send
        TransactionProposal(transactionProposalRequest);

        for (ProposalResponse response : invokePropResp) {
            if (response.getStatus() == Status.SUCCESS) {
                out("Successful transaction proposal response Txid:
                %s from peer %s", response.getTransactionID(), response.getPeer().getName());
                successful.add(response);
            } else {
                out("fiale %s ", "dddd");
            }
        }

        return channel.sendTransaction(successful,user);

    } catch (Exception e) {

        throw new CompletionException(e);

    }

}
```

调用代码如下:

```
ChaincodeID chaincodeID = ChaincodeID.newBuilder().setName("cc_endfinlshed").
build();
sendTranstion(hfclient, testchannel, chaincodeID, admin).thenApply(transactionEvent
```

```

-> {

    String tranid = transactionEvent.getTransactionID();
    System.out.println(" ==== " +tranid);

    return null;

}).exceptionally( e -> {
    return null;
});

```

### 3. Fabric Java SDK 如何进行背书

在 Java 中进行背书操作是非常简单的，只需要在执行 invoke 方法之前，添加需要背书的 Peer 节点即可。相关点如下：

```

Peer peerendorse = hfclient.newPeer( "peer0", "grpc://172.16.10.188:7051");
channel.addPeer( peerendorse );

```

上述代码中还增加了一个背书节点，该背书节点的 IP 地址：172.16.10.188，端口号为：7051。

背书节点必须安装和目标节点一样的 Chaincode。

### 8.3.3 Fabric Java SDK 中 TLS 的设置

如果 Orderer 节点和 Peer 节点都启用了 TLS 模式，那么客户端也需要进行相关的设置，Java 中设置 TLS 的代码样例如下所示：

#### 1. Orderer 节点的设置

```

Properties peerproperties = new Properties();
peerproperties.put("pemFile", "/project/opt_fabric/fabricconfig/crypto-config/
peerOrganizations/org1.robertfabrictest.com/peers/peer0.org1.robertfabrictest.com/
tls/ca.crt");
Peer peer = hfclient.newPeer( "peer0", "grpcs://192.168.23.212:7051",peerproper
ties);

```

#### 2. Peer 节点的设置

```

Properties orderpeerproperties = new Properties();
orderpeerproperties.put("pemFile", "/project/opt_fabric/fabricconfig/
crypto-config/ordererOrganizations/robertfabrictest.com/orderers/orderer.
robertfabrictest.com/tls/ca.crt");
Orderer order = hfclient.newOrderer( "orderer" , "grpcs://192.168.23.212:7050",
orderpeerproperties );

```

设置 TLS 之后，Orderer 节点和 Peer 节点中的访问路径中应该将 grpc 替换成 grpcs。

## 8.4 Fabric Go SDK

Golang 是 Fabric 原生的开发语言，Fabric 和 Fabric-ca 都是用 Golang 开发的，Fabric 的 Chaincode 也是采用 Golang 开发的。从这里可以开出 Fabric、Golang SDK 应该速度最快并且支持最完善的 SDK。在本章中我们将介绍如何使用 Golang 访问 Fabric。

### 8.4.1 Fabric Golang 的安装

Fabric Golang SDK 的安装非常简单，执行如下命令即可

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric-sdk-go
```

// 安装相关的依赖包

```
go get -u github.com/cloudflare/cfssl
go get -u github.com/miekg/pkcs11
go get -u github.com/miekg/pkcs11
go get -u github.com/mitchellh/mapstructure
go get -u github.com/pkg/errors
go get -u github.com/spf13/viper
go get -u golang.org/x/crypto/sha3
go get -u golang.org/x/net/context
go get -u golang.org/x/sync/errgroup
go get -u google.golang.org/grpc
```

### 8.4.2 创建配置文件

Fabric Golang SDK 需要依赖一个配置文件，该配置文件采用 YAML 的格式。配置的风格如下：

```
name: "robert fabric test"
x-loggingLevel: info
client:
  organization: Org1
  logging:
    level: debug
  cryptoconfig:
    path: /project/opt_fabric/fabricconfig/crypto-config
  credentialStore:
    path: "/tmp/hfc-kvs"
  cryptoStore:
    path: /tmp/msp
  BCCSP:
    security:
      enabled: true
    default:
```

```

    provider: "SW"
    hashAlgorithm: "SHA2"
    softVerify: true
    ephemeral: false
    level: 256
organizations:
  Org1:
    mspid: Org1MSP
    cryptoPath: peerOrganizations/org1.robertfabrictest.com/users/{userName}@
org1.robertfabrictest.com/msp
    peers:
      - peer0.org1.robertfabrictest.com
    certificateAuthorities:
      - ca-org1
    adminPrivateKey:
      pem: "/project/opt_fabric/fabricconfig/crypto-config/peerOrganizations/
org1.robertfabrictest.com/users/Admin@org1.robertfabrictest.com/msp/keystore/4cb0b
17fdc61d192984ceb600c90adc4420b5ecec13fe1bc6bc3752ece187e6_sk"
      signedCert:
        path: "/project/opt_fabric/fabricconfig/crypto-config/peerOrganizations/
org1.robertfabrictest.com/users/Admin@org1.robertfabrictest.com/msp/signcerts/
Admin@org1.robertfabrictest.com-cert.pem"
    certificateAuthorities:
      ca-org1:
        url: http://192.168.23.212:7054
        httpOptions:
          verify: true
        registrar:
          enrollId: admin
          enrollSecret: adminpw
        caName: ca-org1

```

上面的配置是运行 Fabric Golang 的最基本的配置，建议在测试环境中使用，完整的配置文件可以参考 [https://github.com/hyperledger/fabric-sdk-go/blob/master/test/fixtures/config/config\\_test.yaml](https://github.com/hyperledger/fabric-sdk-go/blob/master/test/fixtures/config/config_test.yaml)

### 8.4.3 一个简单的 Golang 访问 Fabric 的例子

示例如下。

```

package main

import (

    fabricapi "github.com/hyperledger/fabric-sdk-go/def/fabapi"
    "github.com/hyperledger/fabric-sdk-go/pkg/errors"
    "github.com/hyperledger/fabric-sdk-go/pkg/fabric-client/orderer"
    "fmt"
    identityImpl "github.com/hyperledger/fabric-sdk-go/pkg/fabric-client/identity"
    "io/ioutil"

```



```

"encoding/hex"
afc "github.com/hyperledger/fabric-sdk-go/api/apifabclient"
"github.com/hyperledger/fabric-sdk-go/api/apitxn"
fab "github.com/hyperledger/fabric-sdk-go/api/apifabclient"
peerapi "github.com/hyperledger/fabric-sdk-go/pkg/fabric-client/peer"
"github.com/hyperledger/fabric-sdk-go/pkg/fabric-client/events"
fabricCAClient "github.com/hyperledger/fabric-sdk-go/pkg/fabric-ca-client"
"github.com/hyperledger/fabric-sdk-go/pkg/config"
"github.com/hyperledger/fabric-sdk-go/pkg/logging"
"github.com/hyperledger/fabric-sdk-go/pkg/logging/deflogger"
)

func main() {

    // 初始化日志系统
    if !logging.IsLoggerInitialized() {
        logging.InitLogger(deflogger.GetLoggingProvider())
    }

    // fabric_ca()
    fabric_local()

}

/**
    通过 fabric-ca-server 获取账号信息，并利用获取的账号访问 Peer 服务器节点
*/
func fabric_ca(){

    // 读取 SDK 配置文件
    sdkOptions := fabricapi.Options(ConfigFile: "../fabricsdk/config_test.yaml",)
    // 创建 SDK 代理
    sdk, _ := fabricapi.NewSDK(sdkOptions)
    countext, _ := sdk.NewContext("Org1MSP")
    user := identityImpl.NewUser("Admin", "Org1MSP");

    // 读取 fabric-ca 的配置文件
    configImpl, _ := config.InitConfig("../fabricsdk/config_test.yaml")
    caClient, _ := fabricCAClient.NewFabricCAClient(configImpl, "Org1")
    key, cert, _ := caClient.Enroll("user88", "peer2wd")

    user.SetEnrollmentCertificate(cert)
    user.SetPrivateKey(key)
    session, _ := sdk.NewSession(countext, user)

    // 创建 fabric 客户端代理
    client, _ := sdk.NewSystemClient(session)

    // 创建通道代理，通道名为 :roberttestchannel12

```

```

channel, _ := client.NewChannel("roberttestchannel12" )

// 创建 Orderer 节点代理
orderer, _ := orderer.NewOrderer("grpc://192.168.23.212:7050", "", "", client.
Config())
channel.AddOrderer(orderer)

// 创建 Peer 节点代理
newpeer, _ := fabricapi.NewPeer("grpc://192.168.23.212:7051", "", "", client.
Config())
channel.AddPeer(newpeer)

// 获取当前通道的信息
blockchaininfo, _ := channel.QueryInfo()
fmt.Println(" the peer block height  %d", blockchaininfo.Height)

}

/**
通过 cryptogen 模块生成的账号访问 Peer 服务器节点
*/
func fabric_local() {

// 读取配置文件
sdkOptions := fabricapi.Options(ConfigFile: "./fabricsdk/config_test.yaml",)

// 创建 SDK 代理
sdk, _ := fabricapi.NewSDK(sdkOptions)
session, _ := sdk.NewPreEnrolledUserSession("org1", "Admin")

// 创建 fabric 客户端代理
client, _ := sdk.NewSystemClient(session)

// 创建通道代理, 通道名为:roberttestchannel12
channel, _ := client.NewChannel("roberttestchannel12")

// 创建 Orderer 节点代理
orderer, _ := orderer.NewOrderer("grpc://192.168.23.212:7050", "", "", client.
Config())
channel.AddOrderer(orderer)

// 创建 Peer 节点代理
peer, _ := fabricapi.NewPeer("grpc://192.168.23.212:7051", "", "", client.Config())
channel.AddPeer(peer)

// 获取当前通道的信息
blockchainInfo, _ := channel.QueryInfo()
fmt.Println(" the peer block height  %d", blockchainInfo.Height)

}

```

由于本书篇幅的限制, 上面的代码并不完整, 对相关的异常没有做处理。因此不建议将相关的代码直接使用在生产系统中。

上述代码中的两个方法——`fabric_ca` 和 `fabric_local` 代表了两种获取账号的方式:

- `fabric_local`: 通过使用 `cryptogen` 模块生成的账号访问 Peer 服务器节点。
- `fabric_ca`: 通过使用 `fabric-ca-server` 授权的账号访问 Peer 服务器节点。

#### 8.4.4 Fabric Golang SDK 其他用法

通过上面的实例代码, 我们简单了解了 Fabric Golang SDK 的概况。下面将介绍 Fabric Golang SDK 其他的一些常用的方法。由于篇幅的限制下面的示例代码我们只给出部分, 如果想要运行这些代码可以把这些代码复制到上面示例中 `fabric_ca` 或者 `fabric_local` 方法的最后。

这里还是按照系统管理和 Chaincode 相关操作两个部分来介绍。

##### 1. Fabric Golang SDK 系统管理相关接口

- 获取当前 Peer 加入了哪些 Channel

```
channels,err := client.QueryChannels(peer)
for _, responsechannel := range channels.Channels{
    fmt.Println(" the channel info is : %s " , responsechannel.ChannelId )
}
```

- 获取当前 Peer 加入的某个 Channel 的区块数

```
blockchaininfo, _ := channel.QueryInfo()
fmt.Println(" the peer block height %d", blockchaininfo.Height)
```

- 根据当前的 Peer 加入的某个 Channel 中的某个区块号, 获取区块的详细信息

```
block, err := channel.QueryBlock(23)
fmt.Println(" The block info : %s " , hex.EncodeToString(block.Header.
PreviousHash))
```

- 根据当前的 Peer 加入的某个 Channel 中的某个区块的哈希值, 获取区块的详细信息

```
let blockinfoobyhash = yield Channel.queryBlockByHash(new Buffer("ec298dc1cd1f0
e0a3f6d6e25b5796e7b5e4d668aeb6ec3a90b4aa6bba7f0c17", "hex"), peer)
console.info( JSON.stringify(blockinfoobyhash) )
```

- 根据当前的 Peer 加入的某个 Channel 中交易的哈希值, 获取交易的详细信息

```
tran , err := channel.QueryTransaction("56f51f9a54fb4755fd68c6c24931234a59340f
7c98308374e9991d276d7d4a96")
fmt.Println(" transcaion info is : %s " , tran.String())
```

- 获取当前 Peer 服务器中状态为 `install` 的 Chaincode 的信息

```

installchaincodes , err := client.QueryInstalledChaincodes(peer)
for _ , responseinstall := range installchaincodes.Chaincodes{
    fmt.Println(" chaincode info is : %s %s " , responseinstall.Version ,
responseinstall.Path )
}

```

## 2. Fabric Golang SDK 中 Chaincode 相关操作相关接口

### ● 调用 Chaincode 的 query 方法

```

targets := peerapi.PeersToTxnProcessors(channel.Peers())
request := apitxn.ChaincodeInvokeRequest{
    Targets:    targets,
    ChaincodeID: "cc_endfinlshed",
    Fcn:        "query",
    Args:       [][]byte{{[]byte("a")}},
}

```

```

queryResponses, err := channel.QueryByChaincode(request)
if err != nil {
    check(err, "QueryByChaincode failed %s")
}

```

```

for _ , parmbytes := range queryResponses{
    fmt.Println(" chaincode query info is : %s " , string(parmbytes))
}

```

### ● 调用一个已经部署的 Chaincode 的 invoke 方法

```

targets := peerapi.PeersToTxnProcessors(channel.Peers())
transientData := make(map[string][]byte)

```

```

request := apitxn.ChaincodeInvokeRequest{
    Targets:    targets,
    Fcn:        "invoke",
    Args:       [][]byte{{[]byte("a")}, []byte("b"), []byte("1")},
    TransientMap: transientData,
    ChaincodeID: "cc_endfinlshed",
}

```

```

transactionProposalResponses, txnID, err := channel.SendTransactionProposal(request)
fmt.Println(" tx id : %s", txnID)
if err != nil {
    check(err, " send transtion error ")
}

```

```

for _ , v := range transactionProposalResponses {
    if v.Err != nil {

```



```

        check(v.Err, "endorser %s failed")
    }

}

tx, err := channel.CreateTransaction(transactionProposalResponses)
transactionResponse, err := channel.SendTransaction(tx)

fmt.Println(" search result  %s " , transactionResponse )

```

#### 8.4.5 Fabric Golang SDK 的背书操作

在 Golang 中进行背书操作是非常简单的,只需要在执行发起交易之前,添加需要背书的 Peer 节点即可。相关代码如下:

```

peerendorse ,err := fabricapi.NewPeer("grpc://172.16.10.188:7051","", "",client.
Config())
channel.AddPeer(peerendorse)

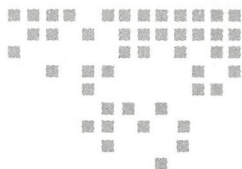
```

上述代码中增加了一个背书节点,该节点的 IP 地址为: 172.16.10.188, 端口为号为: 7051。背书节点必须安装和目标节点一样的 Chaincode。

背书操作虽然简单,但是在每个语言的 SDK 中都进行了单独的提示,这是一个非常重要的功能,但是相关的 SDK 都做了非常好的封装所以通过这些 SDK 进行背书操作还是比较简单的。从这一点可以看出使用 Fabric 相关语言的 SDK 进行开发,可以简化我们的开发难度,大大提高开发效率。

## 8.5 本章小结

本章主要介绍了 Fabric 的 Grpc 接口,熟悉本章内容有助于了解客户端程序是如何同 Fabric 进行交互的,本章内容在 Fabric 的项目中非常重要。



## Fabric 系统架构设计

区块链的概念比较新，但是区块链的技术都是已经存在的技术。新概念和现有技术结合之后会产生一些新的特性，这些特性决定了采用区块链技术的系统架构和现有的系统架构之间既有联系又有区别。基于区块链技术的系统在进行技术架构设计的时候除了需要考虑到现有技术架构的特点之外，还需要考虑到区块链技术的自身特点。

Fabric 作为一个典型的区块链技术框架，其系统架构设计也需要遵循上述特点。本章内容将介绍如何对基于 Fabric 技术的系统进行系统架构设计。

### 9.1 Fabric 架构中的组织规划

组织是 Fabric 中非常重要的概念，所有的 Peer 节点、用户账号都必须属于同一个组织。Fabric 中的组织在现实世界中可以是一个公司、一个企业，或者是一个协会。在 Fabric 中，组织通常是具有承担数据信用责任的区块链系统参与方。在设计一个 Fabric 系统的时候，第一步要做的事情就是收集系统的参与者，然后从这些参与者中选出相关的组织。组织被选出之后需要确认组织的管理方式，组织的管理方式是指组织在遇到问题的时候协作的方式，这些工作更多的是线下的协调工作，但是经过协调之后组织之间必须达成共识来确认一种组织管理方式。

综上所述，我们认为在 Fabric 架构中的组织规划工作实际上需要完成两件事情：确认组织和制定组织之间的管理方式。

### 9.1.1 确认组织

目前在设计 Fabric 系统的时候, 如何确认系统的参与者是否可以成为一个组织还没有统一的标准。根据我们在 Fabric 项目中的实施经验, 我们给出以下条件供读者参考。我们认为一个参与者如果想要成为 Fabric 系统中的组织, 必须具备以下条件:

- 是否对区块链中的数据具有有效性检查的权力
- 是否具有独立发展下属成员的权力和资格
- 是否对系统的核心业务不可或缺

具有以上条件的系统参与者都可以成为系统的组织。参与者成为组织后, 会有自己的组织编号、域名、证书等信息。成为 Fabric 系统组织的参与者将有机会参与到 Channel 的维护工作, 同时成为 Fabric 系统的组织之后, 相关的参与者将会拥有自己的认证服务器, 通过这些认证服务器可以随意添加组织内部节点。由此可见在系统参与之中, 确认组织是 Fabric 系统架构设计的重要组成部分。

### 9.1.2 组织的管理方式

在组织确认后, 接下来的问题就是要确认系统对组织的管理方式, 所有管理方式中最重要的就是新组织的加入问题。在系统运行的过程中组织不是一成不变的, 由于业务的变化会有新的组织加入到系统中或者将已有的组织从系统中删除, 针对这些操作需要制定相关的规则。在 Fabric 系统的架构中, 协调组织的管理方式也是系统架构设计的一部分。我们知道 Fabric 是个联盟链, 联盟链的特点是不是所有的组织都可以随意加入到区块链中。如果想要加入到区块链中, 需要一定的准入机制。在 Fabric 系统中, 如果新的组织想要加入区块链中, 需要所有组织的一致同意才可以加入。如果区块链的现有组织中有任何一个组织不同意新组织的加入, 那么新组织就不能加入到区块链中。如果需要将现有的组织中的一个删除, 也需要所有组织的签名。

在实际的操作中, 对组织的管理实际上就是对组织所拥有的证书的管理。在前面的内容中我们介绍了 Fabric 系统架构设计的第一步是确认组织, 组织确认完成之后会生成组织所需要的相关的证书, 证书是组织在整个区块链中的身份标识。举个例子, 如果系统中的所有组织都允许新组织的加入, 那么需要所有组织用其特有的证书对相关的操作进行签名后才能正确完成这些操作。通常对 Fabric 的组织证书的管理方式可以采用以下两种模式: 联盟式管理和集权式管理。

#### 1. 联盟式管理

联盟式管理是 Fabric 设计的初衷, 如果对组织的证书采用联盟式管理的方式, 那么系统初始化完成之后需要将生成的各个组织的相关证书(包括组织的证书、组织用户的证书、



组织中所有节点的证书)全部提供给相关组织(或者这些证书由相关的组织自行生成,以后这些证书由所属组织独立维护。组织的证书采用联盟式管理的好处是所有参与的组织之间是对等的,可以共同参与整个联盟链的管理。但是坏处也是明显的,整个联盟链中没有一个统一的管理机构,如果有组织之间发生冲突,那么只能通过协商的方式处理,整个系统没有仲裁结构。举个例子,如果需要删除区块链中的某个组织,必须获取包括被删除组织在内的所有组织的同意才能完成删除操作。这个时候如果被删除组织不同意,那么无法完成删除操作。由此可见采用联盟式管理必须建立在组织之间互相熟悉的基础之上。

## 2. 集权式管理

正是由于联盟式管理出现了管理不统一的问题,所以才存在着另外一种管理方式——集权式管理。在集权式管理中,整个区块链中存在一个中心组织,该中心组织统一管理区块链中的所有组织。区块链中所组织的管理员账号的证书将被收回,由中心组织统一管理。如果需要增加新的组织或者删除已经存在的组织,只有中心组织同意即可完成相关的操作。

联盟式管理和集权式管理哪个更好呢?毋庸置疑是联盟式管理,因为联盟式管理更符合区块链的哲学。但是我们在具体项目中需要考虑到历史遗留的问题,具体选择哪种方式需要根据项目的具体情况来决定。

## 9.2 Fabric 系统的结构

在确定了区块链中的组织和组织之间的管理方式之后,接下来要做的事情是设计 Fabric 的系统结构。在本书前面的章节中我们知道,一个完整的 Fabric 系统中由 Peer、Orderer、Fabric-ca、Kafka 等模块组成。其中 Peer、Orderer 是 Fabric 的核心模块, Fabric-ca-server 是 Fabric 的扩展模块,负责动态生成 Fabric 的账号,而 Kafka 是一个独立的消息队列,负责存储 Fabric 的打包数据。这些系统的关系如图 9-1 所示。

通过图 9-1 我们可以发现, Fabric 系统是通过组织来划分的,每个组织内都包含承担不同功能的 Peer 节点,同时每个组织都有自己对应的 Fabric-ca 服务器。所有的组织共用一个统一的 Orderer 集群。因此在设计 Fabric 系统结构的时候需要对每个模块的内部属性和模块之间的联系方式进行一定的规划。具体可以通过逻辑层和物理层两个方面考虑。

### 9.2.1 Fabric 系统的逻辑结构

逻辑结构主要是描述 Fabric 系统中组织与组织以及组织内的关系。逻辑结构是整个系统业务模式的直观反映,通过逻辑结构可以大致了解系统的范围,为后面的设计打下基础。在设计 Fabric 系统逻辑结构的时候,可以参考图 9-1,通过绘制该系统架构图,可以了解整个系统逻辑结构,确定系统设计的范围。在设计 Fabric 系统逻辑结构的时候,我们建议读



者以图 9-1 为例，绘制出整个系统的架构图，并以此作为后续设计工作的基础。

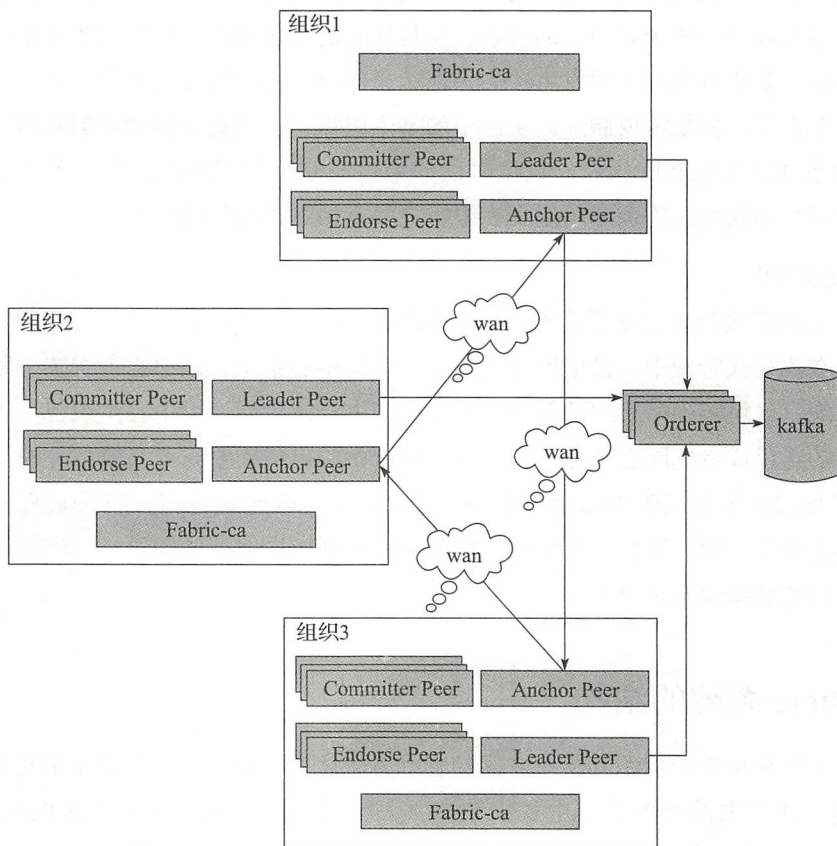


图 9-1 Fabric 系统架构图

在系统的逻辑结构被确认之后，在 Fabric 系统的逻辑结构设计过程中还需要确认各个组织所使用的域名。在 Fabric 的初始化配置中很多地方需要使用到相关的域名，域名是 Fabric 系统中很多模块（比如 Peer、Orderer 等）的访问路径，因此需要在逻辑设计阶段确认相关的域名。我们建议以表 9-1 为例，来制作组织域名匹配表。

表 9-1 Fabric 系统组织和域名匹配表

组织名称	对应域名
org1	qklszznorg1.com
org2	qklszznorg2.com
org3	qklszznorg3.com

上述表格建立之后，需要经过区块链中所有组织的集体确认后方可使用。



注意 上述域名最好是经过备案后使用。

### 9.2.2 Fabric 系统的物理结构

在设计 Fabric 系统物理结构的时候需要注意配置文件的路径和访问域名。因为这些配置文件的路径有些是会被写到系统的创始块中的，一旦写入到创始块中，系统在其他服务器上部署的时候必须保证配置文件的路径和创始块中的路径是一致的，否则系统会出现错误。同样，对于域名配置也是一样，一些域名一旦被配置之后，相应的域名信息就会被写入到创始块中，后面修改非常困难。因此在系统正式发布时需要使用正规的域名（如果是在国内，使用的域名必须是经过备案的）。

目前 Fabric 系统中会写入到创始块中的配置信息是 configtxgen 模块的配置文件，我们截取部分配置代码如下所示：

```
- &Org1
    # DefaultOrg defines the organization which is used in the sampleconfig
    # of the fabric.git development environment
    Name: Org1MSP

    # ID to load the MSP definition as
    ID: Org1MSP

    MSPDir: /opt/hyperledger/fabricconfig/crypto-config/peerOrganizations/
org1.robertfabrictest.com/msp

    AnchorPeers:
        # AnchorPeers defines the location of peers which can be used
        # for cross org gossip communication. Note, this value is only
        # encoded in the genesis block in the Application section context
        - Host: peer0.org1.robertfabrictest.com
          Port: 7051
```

在上述配置示例中，MSPDir 属性和 AnchorPeers 属性都会被写入到创始块中。

## 9.3 Fabric 中 Channel 的设计

Fabric 的数据存储结构被设计成多账本体系，每一个账本在 Fabric 中被称为 Channel，加入 Channel 中的每个 Peer 节点都是对等的，也就是说同一个 Channel 中的所有 Peer 节点都保存一份相同的数据。但是 Fabric 目前并没有提供分布式存储的解决方案，这导致了加入了同一 Channel 的 Peer 节点服务器中都会重复存储相同的数据。这种存储结构在数据量非常大的时候会影响 Peer 节点读取数据的性能。

基于 Fabric 这样的存储特性，在设计 Fabric 系统的时候需要对 Channel 存储方式进行相关的设计。Fabric 对 Channel 的存储方式进行设计的内容主要是：首先对 Channel 的大小进行评估，如果单个 Channel 的数据量或者数据的存储空间都非常大，那么可以将一个非常大的 Channel 拆分成若干个相对较小的 Channel。

我们通过一个例子来说明如何拆分 Channel。假设在 Fabric 系统中我们有这样一个 Channel，该 Channel 中包含的数据可能超过 10 亿条，数据文件的大小可能超过 500G。如果按照 Fabric 现有的设计方式，Channel 和节点的分布如图 9-2 所示：

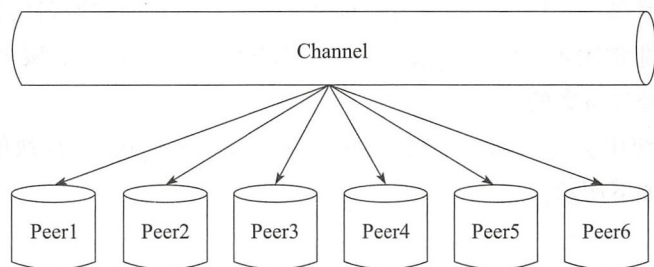


图 9-2 Fabric 大 Channel 分布示意图

根据图 9-2 所示，以后的 Peer 节点都要存在大约 500G 的数据，这显然对单个 Peer 节点造成了比较大的压力。这个时候我们可以采取对 Channel 进行分割，我们将每个 Channel 的数据限制在 2000 万条，经过分割之后的 Channel 结构如图 9-3 所示：

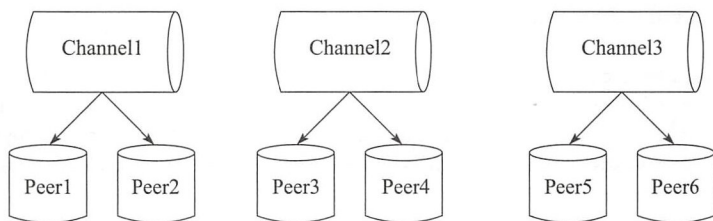


图 9-3 Fabric 大 Channel 分布示意图

通过图 9-3 可见，经过分割处理之后的 Channel 体积缩小到原来的三分之一。对 Channel 进行分割处理还有一个好处就是，如果数据继续增长只需要增加一个新的 Channel 即可，这样理论上可以应对数据的无限增长。

## 9.4 Chaincode

在本书的第 7 章中我们详细地介绍了 Chaincode 原理、开发和使用等基本功能和操作，这里不再复述。我们知道 Chaincode 是 Fabric 重要的组成部分，对 Channel 进行的任何操作



都是通过 Chaincode 进行的。在设计 Fabric 系统的时候针对 Chaincode 的特性需要关注以下几个问题。

(1) Chaincode 只作为数据的加工者存在，不要存储中间状态。

同一份 Chaincode 通常会部署在多个 Peer 节点中。因此在每个机器的状态可能是不一样的，因此在 Chaincode 的代码中，尽量不要通过本机的属性（时间戳、硬盘编号）生成诸如随机数等包含节点特定属性相关的数据。把 Chaincode 当成一个数据处理状态机，只进行数据的校验、读取、存储等基本操作。

(2) 在 Chaincode 中避免进行 CPU 和内存密集型运算。

Chaincode 是运行在容器中的，每个 Chaincode 分配的资源受到限制，因此如果在 Chaincode 中进行大 CPU 或者大内存的运算可能导致容器的崩溃。在设计 Chaincode 代码的时候需要考虑到这一特点，我们建议在将 Chaincode 部署到生产系统之前首先进行相关的压力测试，以便获取当前 Chaincode 在运行过程中 CPU 和内存等相关系统参数的阈值，进而对 Chaincode 代码进行优化。

## 9.5 数据访问层

当需要在 Fabric 中对系统进行管理操作或者调用 Chaincode 的时候通常可以使用两种方法：CLI 命令行工具和 Fabric SDK 调用。CLI 命令行工具需要获取 Peer 节点所在主机的操作系统的账号，然后登录操作系统之后使用，这种方式通常被系统管理员采用。如果需要在业务系统中通过代码的方式同 Fabric 进行交互，只能通过 Fabric 提供的相关语言的 SDK 编写相应的代码和 Fabric 进行交换。在设计这些访问代码的时候通常会采取两种方式：嵌入式和中间件式。这两种代码的设计方式如图 9-4 所示：

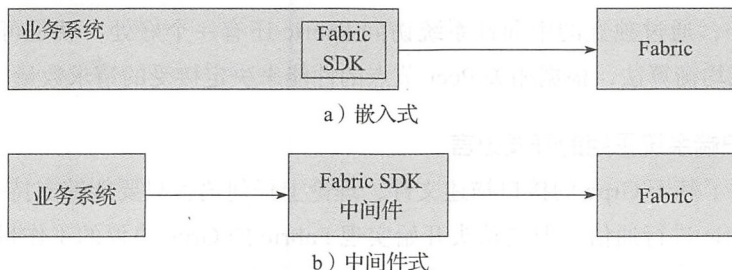


图 9-4 业务系统访问 Fabric 代码设计示意图

在图 9-4 所示的两种方法中，我们建议采用方式 b。在 Fabric 系统中设计一个独立的中间层封装对应 Fabric 常用操作，将这些操作采用通用接口的方式提供给业务系统调用。我们认为这种设计方式有以下好处：



### 1. 解决 Fabric 记录状态返回不实时的问题

Fabric 是基于区块链的分布式总账系统, 所有的数据存放于区块链中。区块链的区块生成是需要时间的, 目前 Fabric 的 Orderer 模块专门负责将交易打包成区块。区块的生成按照时间或者交易的数量进行打包, 比如可以设置每一分钟打包一个区块或者每个 1000 条记录打包一个区块。无论采用哪种方式, 区块链的生成都是需要一定时间的。因此 Fabric 系统中是无法立即响应数据操作状态, 如果提交一笔交易之后则需要一段时间才能获取该交易的操作结果。如果采用将 Fabric 相关的操作集成在一个中间件中, 遇到类似场景的时候可以数据处理完成之后, 通过中间件将处理结果推送给业务系统进行后续处理。这样可以有效解决业务系统的等待问题, 同时还可以降低业务系统的开发难度。

### 2. 有利于系统进行负载均衡

Fabric 并不是一个可以承载大并发读写的数据系统, 当系统请求的并发比较多时候 Fabric 会产生阻塞, 遇到这种情况通常可以采用增加 Peer 节点的方式来减轻单个节点的负载。

举个例子, 在 Fabric 系统中某个 Channel1 中存在一个数据访问节点 Peer1, 该节点专门负责接收外部系统的访问请求。随着系统运行外部系统的请求数量会逐步增加, 这个时候 Peer1 节点可能无法满足需求因此需要增加一个新的节点分担 Peer1 节点的负载, 我们将新增加的节点称为 Peer2。增加的 Peer2 节点加入 Channel1 之后, 系统会自动将 Channel1 中的数据部署到 Peer2 中。然后在 Peer2 中部署和 Peer1 中同样的 Chaincode, 这样 Peer2 就可以提供和 Peer1 一样的功能。通过这样的方式, Fabric 系统能够简单的对系统进行横向扩展。但是这种特性对业务系统访问 Fabric 的方式提出了相关的要求。假设在业务系统中编写访问 Fabric 的代码, 这时如果 Fabric 增加了一个新的访问节点, 业务系统可能需要进行重启、打包等维护工作。这样可能会降低业务系统的稳定性。如果将对 Fabric 的操作封装在一个独立的中间件中, 可以在不影响业务系统的情况下直接更新中间件系统即可完成相关的升级工作。除此之外, 通过独立的中间件系统访问 Fabric 还有一个好处, 就是可以在中间件中集成相关的负载均衡算法, 根据相关 Peer 节点的性能来决定接受的请求数量。

### 3. 隔离客户端系统采用的开发语言

Fabric 提供了基于 Grpc 的接口描述文件, 理论上任何语言只要能够支持 Grpc 协议都能够很好地和 Fabric 进行通信。但是从头开始实现 Fabric 的 Grpc 协议的工作量非常大, 而且要进行很多测试工作。目前 Hyperledger 项目已经实现了 Nodejs、Go、Java、Python 这四个语言的 SDK, 应用这四种语言开发的系统可以非常容易的和 Fabric 进行通信。但是常用的编程语言远不止这些, 开发业务系统的编程也不会局限于这四种语言。比如, 业务系统采用的开发语言是 PHP, 这就需要从头开始用 PHP 根据 Fabric 提供的 Grpc 的描述文件生成相关的代码, 这个过程比较繁琐而且容易出错。这时候如果采用 Nodejs、Go、Java、Python

这四种语言中的一种开发一个通用的中间件，然后通过一些常用的协议（比如 JSON-RPC）暴露接口，这样能够屏蔽由于业务系统的编程语言多样性带来的问题。

## 9.6 历史遗留系统的兼容

在采用 Fabric 架构的系统中我们常常面对这样的场景：需要将现有的系统和 Fabric 系统对接。比较常见的是溯源系统，这样的系统在引入区块链技术之前业务方已经有一个基于传统技术架构的系统，该系统已经运行了一段时间并积累了大量的数据，现在为了提高数据价值准备将所有的历史溯源数据存放到区块链中，并且在系统改造完成之后，新产生的溯源数据也需要存放到区块链中。在实际工作中这样的业务场景和需求是经常会遇见的。

在上述溯源的业务场景中，最核心的问题是历史遗留系统和 Fabric 的结合。针对这样的业务需求，我们建议从账号同步和数据融合这两个方面来进行考虑。

### 1. 业务系统的历史账号和 Fabric 账号的同步

如果业务系统中存在历史遗留系统，首先需要仔细分析历史遗留系统中的账号体系。分析可以从账号是否会继续增加和现有账号的数量这两个维度来考虑历史遗留系统的账号体系和 Fabric 账号体系的融合。

#### （1）历史遗留系统中账号体系中的账号是固定不变的

在这种情况下如果账号的数量比较少，可以通过 Fabric 的 cryptogen 模块生成相应数量的账号信息，然后将生成的账号文件和历史遗留系统的账号进行绑定。如果账号的数量比较多，那么可以借助 Fabric-ca-server 来对这些账号进行管理，通过 Fabric-ca-server 提供的 RESTAPI 接口可以将历史遗留系统的账号导入到 Fabric-ca-server 中。在调用 Fabric-ca-server 提供的 RESTAPI 导入数据时，每次调用成功都会返回当前创建账号的证书等信息，通过这些信息可以完成与历史遗留系统账号体系的绑定。

#### （2）历史遗留系统的账号会持续增加

如果是这种情况，那么只能采取 Fabric-ca-server 来管理账号，通过 Fabric-ca-server 提供的 RESTAPI 接口将历史遗留系统的现有账号导入到 Fabric-ca-server 中完成绑定。对于新增的账号，每次创建账号之后，调用 Fabric-ca-server 的接口生成新的账号并完成绑定。



关于 Fabric 的账号以及 Fabric-ca-server 的相关内容请参考本书的第六章。

### 2. 业务系统历史遗留数据和 Fabric 的融合

关于历史遗留系统的已有数据如何保存到区块链中是一个比较麻烦的问题。在处理这



个问题的时候会有一个矛盾。由于历史遗留数据是发生在过去的,但是区块链中每条交易都会在系统级别记录一条数据的时间戳。这个时间是客户端无法修改的并将永远保存。这个时候如果进行数据初始化就会发生数据存储时间不一致的问题,即历史数据的系统时间戳和业务时间戳是不一致。这可能会导致数据有效性受到质疑。这个时候区块链技术已经无法解决这个问题了,只有联盟链的所有参与方达成共识,统一接受这样的事实才可以。针对这种问题我们推荐两种解决方案供大家参考,这两种方案分别是:时间戳截取法和 Channel 截取法。

#### (1) 时间戳截取法

时间戳截取法是通过设定某个特定的时间点,在该时间点之前的数据为历史数据,时间点之后的数据为新数据。当系统中需要用数据的时间戳的时候,对于时间节点之前的数据以存在交易记录中的历史时间为准,对于时间点之后的数据以系统时间戳为准。

#### (2) Channel 截取法

在使用 Channel 截取法的时候,将历史数据存储在新的通道中。当需要使用历史数据的时候从特定的通道中获取相关的数据,如果需要使用新生成的数据则从新创建的 Channel 中获取数据。

上述两种方法都需要在历史数据保存到区块链中的时候将历史数据中的时间戳一并保存到区块链中。



**注意** 在设计区块链系统的时候,技术不是万能的,很多时候需要所有的参与者达成共识才能有效地解决问题。人是设计区块链系统必须充分考虑的一个因素。区块链提出了一个去中心化的理想模型,但是直接用来改造现实社会是有问题的。在设计基于区块链技术的系统的时候,需要在理想和现实之间达成妥协。

## 9.7 Fabric 系统的维护和管理

区块链系统和传统的以数据库为核心的系统有很多不一样的地方,因此在对系统进行维护的时候需要着重注意以下三个问题:

### 1. 数据备份

我们知道 Fabric 系统是一个区块链系统,而区块链本身具有分布式数据库的特性,整个系统强调最终一致性,在某个时间点很难确认那个节点的数据是最完整的,所以传统数据库系统中采用的导出导入的备份和恢复方式不适合 Fabric 系统。但是可以利用区块链系统分布式数据库的特性,通过增加节点的方式进行备份。比如在 Fabric 系统中设置专门的数据备份 Peer 节点,该节点在加入相关的 Channel 之后不接受任何外部程序的访问请求,专

门作为备份节点使用。

## 2. 系统升级

Fabric 的一个开源系统，目前处于快速发展阶段，代码的变化比较频繁。在现阶段采用 Fabric 系统开发系统，一定要注意系统升级之后数据格式的兼容性。比如 Fabric0.6 版本和 Fabric1.0 版本的数据是不兼容的。我们建议在开发基于 Fabric 系统的时候一定要做好相关的操作日志，如果出现升级之后数据格式和系统版本不兼容的情况可以在联盟所有参与方共同协商并且都认可的情况下进行相关的数据恢复操作。

## 3. Orderer 节点的多机备份

Orderer 模块是 Fabric 系统中的核心模块，Orderer 模块主要负责数据的排序和打包，一旦 Orderer 节点出现问题，整个区块链系统都会出现问题。因此 Orderer 节点应该避免单个节点。可以部署几个拥有相同创始块的 Orderer 节点，然后把这些节点指向同一个 Kafka 集群。

# 9.8 本章小结

Fabric 作为区块链技术框架有其自身的特点，在开发基于 Fabric 技术框架的系统中的时候不能在不经考虑的情况下随意使用，基于 Fabric 技术框架的系统也是需要设计的，本章主要介绍 Fabric 系统在设计是需要注意的事项。良好的架构设计是系统成功的一半，区块链系统也是需要进行架构设计。



## Fabric 开发实战：开发流程与实例详解

区块链技术是利用现有技术进行精巧组合之后的新技术框架，因此基于区块链技术的系统和采用传统技术的系统有很多区别。这些区别导致区块链系统在系统设计和开发流程上面有很多独特的地方。Fabric 作为一个典型的企业级区块链平台，也存在这样的特性，因此基于 Fabric 项目的开发流程中存在一些和传统项目不一样的地方。

本书特意将 Fabric 项目的开发流程单独作为一个章节，通过一个复杂的例子向读者介绍如何实施一个 Fabric 项目。



**注意** 阅读本章的过程中如果读者想要进行相关的实例操作，请勿首先仔细阅读本书的第 4 章并且正确完成其中的所有的操作。

### 10.1 Fabric 项目的开发流程

在本节中，我们将模拟建立一个有 3 个组织的 Fabric 系统，在该系统中每个组织都有自己独立的 CA 服务器，每个组织内都有 Peer 节点和用户，所有组织共用一个具有负载均衡功能的 Orderer 节点群，这些 Orderer 节点可以任意地扩展。在创建相关配置文件之前我们需要确定以下参数。

#### 1. 统一的域名

在 Fabric 项目开始前首先需要选择一个域名，因为 Fabric 是个联盟链，在项目开始之

初要充分考虑到未来可能需要引入的新组织。在选择域名的时候尽量选择已经完成备案的域名。本例中采用了一个测试域名：qklszzn.com，这个域名只是测试使用，如果在正式项目中应用请选择经过备案的域名。

## 2. 演示项目的成员结构

演示项目的成员是指整个 Fabric 区块链系统中参与的组织，以及这些组织中包含的节点。我们假设在即将演示的 Fabric 系统包含以下组织，如表 10-1 所示：

表 10-1 Fabric 演示系统组织标识说明对应表

标识	说明
bc_org1	模拟组织 1
bc_org2	模拟组织 2
bc_org3	模拟组织 3

演示系统初始化的时候有三个组织，后面我们会模拟如何在系统中增加一个新的组织。

## 3. 组织中包含的初始节点

演示项目的每个组织都会包含若干个节点，如表 10-2 所示：

表 10-2 Fabric 演示系统组织节点对应表

组织表示符号	节点标识符	说明
bc_org1	bc_org1_peer1	组织 bc_org1 的主节点
bc_org1	bc_org1_peer2	组织 bc_org1 的备份节点
bc_org1	bc_org1_peer3	组织 bc_org1 的背书节点
bc_org2	bc_org2_peer1	组织 bc_org2 的主节点
bc_org2	bc_org2_peer2	组织 bc_org2 的备份节点
bc_org2	bc_org2_peer3	组织 bc_org2 的背书节点
bc_org3	bc_org3_peer1	组织 bc_org3 的主节点
bc_org3	bc_org3_peer2	组织 bc_org3 的备份节点
bc_org3	bc_org3_peer3	组织 bc_org3 的背书节点

每个组织的初始化节点不是很多，新增加的节点通过 CA 服务器发布证书。初始化的时候每个组织都会包含 3 个节点：主节点、备份节点、背书节点。

- 主节点：默认对外发布的节点，客户端默认通过主节点访问相关的信息，主节点不参与背书过程。
- 背书节点：专门提供背书服务的节点。
- 备份节点：提供主节点和背书节点的功能，当主节点或者背书节点出现问题时临时替代它们的功能。

## 4. 系统中的 Channel

演示系统中需要创建一些 Channel，这些 Channel 和组织之间的关系如下表所示：

表 10-3 Fabric 演示系统组织和 Channel 对应关系表

Channel 标示符	Channel 中包含的组织
bc_channel1	包含组织 bc_rog1 和 bc_org2
bc_channel2	包含组织 bc_rog1 和 bc_org3
bc_channel3	包含组织 bc_rog2 和 bc_org3
bc_channel4	包含组织 bc_rog1、bc_org2 和 bc_org2

这是系统初始化时需要创建的通道，在新增加组织后会演示如何在现有的通道中增加新的组织。

5. Order 的共识节点

为了模拟生产系统，本测试的共识方法选取 Kafka 的方式。Kafka 服务器的集群和配置方法可以参考 Kafka 的相关文档。

上述示例可以用图 10-1 来表示本次示例需要达到的效果。

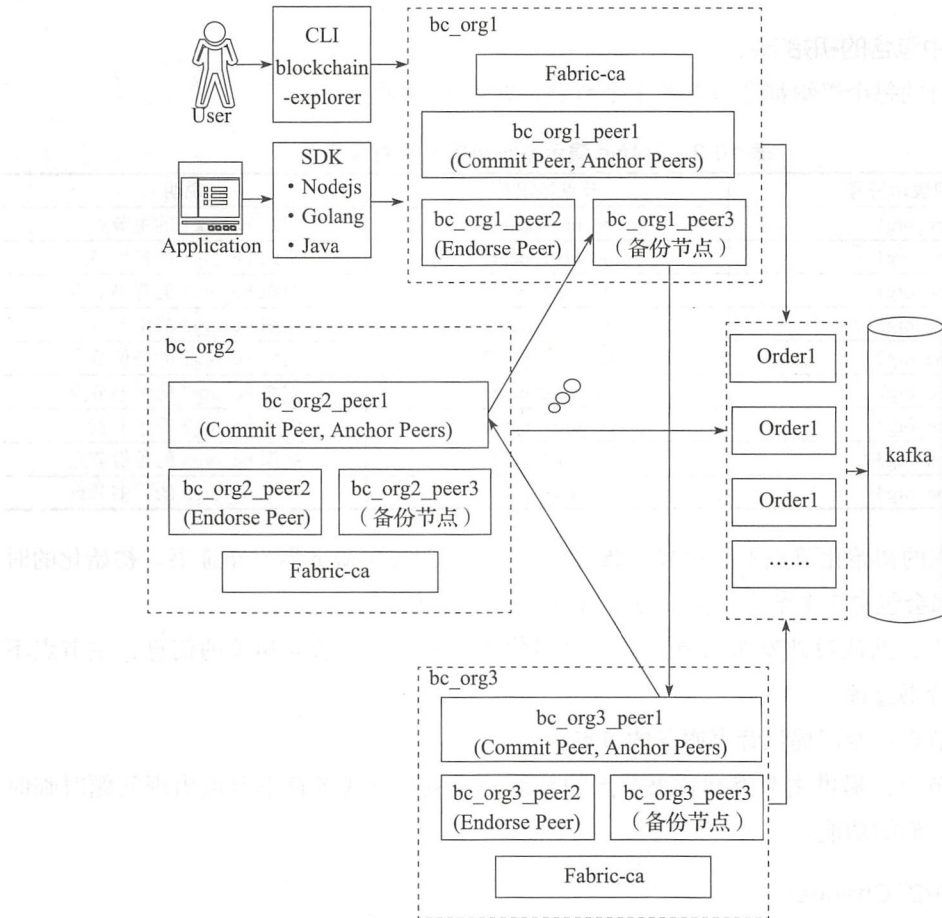


图 10-1 Fabric 模拟系统组织架构图

## 10.2 Fabric 项目开发实例详解

Fabric 的编译、配置、账号管理、Chaincode 等部分的内容在前面已经介绍过，本节实例是在前面这些内容的基础上进行的，希望读者在充分理解前面章节的基础上阅读本章。在本次示例开始前，我们会根据相关项目的流程给出一个流程图来说明 Fabric 项目的实施过程。在流程的每一个节点中，我们会列出本节点采用的工具、配置文件，输出文件，以及需要在后面引起注意的域名和文件目录。



**注意** 作者在运营区块链技术论坛和 Fabric 实际项目中发现，域名和相关证书文件的路径是很容易出错的，所以我们会列出每一步相关的域名和配置文件供读者参考。同时在开始下面的流程之前请先正确安装并启动 Kafka 服务器。

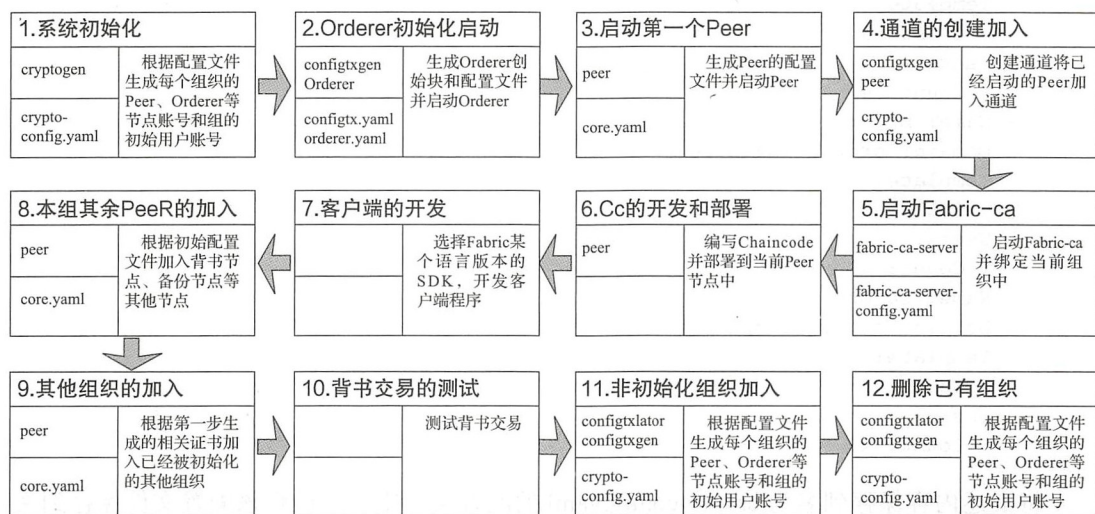


图 10-2 Fabric 项目开发流程图

上图中每个步骤分为 3 个部分，分别表示当前步骤需要的 Fabric 模块、配置文件以及需要完成的工作。

### 10.2.1 系统初始化

Fabric 在开始之前，我们建议首先根据项目的需求，把整个项目中可能会涉及的参与方（组织）和参与方大概的规模（节点和用户账号数）罗列出来。10.1 节的内容就是一个简单的项目分析结果清单。我们建议读者在 Fabric 项目开始之前可以参考 10.1 节中的图例，将相



关的组织 and 组织的规模画出来先分析一下, 然后也列出类似的表格, 最后根据这些表格的内容开始编写配置文件。在项目演示开始之前, 我们首先创建一个文件夹, 这个文件夹用来存放所有的配置文件以及后面的演示过程中生成的相关文件。本例中的文件夹如下所示:

```
mkdir -p /var/qklszzn
```

目录创建完成后, 以 10.1 节表格为参考开始编写 `cryptogen` 模块需要使用的配置文件。配置文件内容如下所示:

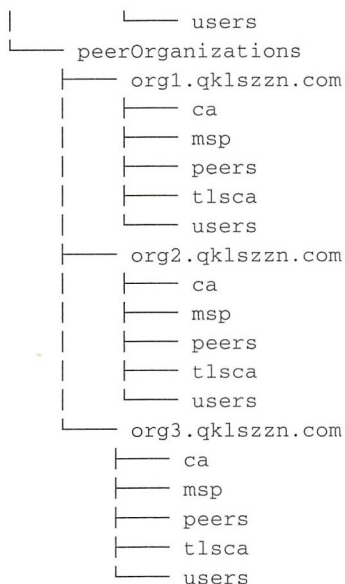
```
OrdererOrgs:
  - Name: Orderer
    Domain: qklszzn.com
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: bc_org1
    Domain: org1.qklszzn.com
    Template:
      Count: 3
    Users:
      Count: 4
  - Name: bc_org2
    Domain: org2.qklszzn.com
    Template:
      Count: 3
    Users:
      Count: 4
  - Name: bc_org3
    Domain: org3.qklszzn.com
    Template:
      Count: 3
    Users:
      Count: 4
```

将上述内容保存到名为 `crypto-config.yaml` 的配置文件中, 然后将该配置文件保存到文件夹 `/var/qklszzn` 中, 最后调用 `cryptogen` 模块生成相关的配置文件, 生成配置的命令如下:

```
cryptogen generate --config=crypto-config.yaml --output ./crypto-config
```

`cryptogen` 模块执行成功之后, 生成的配置文件会保存到当前目录下面的 `crypto-config` 文件夹中。`crypto-config` 中的内容如下所示:

```
|— ordererOrganizations
|   |— qklszzn.com
|       |— ca
|       |— msp
|       |— orderers
|       |— tlsca
```



## 10.2.2 Orderer 节点的初始化和启动

Orderer 模块启动之前需要生成一个创始块文件和 Orderer 模块所需的配置文件。创始块是 Fabric 的第一个区块，主要存储相关的配置信息。

### 1. Fabric 创始块的生成

创始块的创建需要通过 configtxgen 模块来生成，configtxgen 模块需要一个配置文件来描述创始块的配置信息。本例中的配置文件的文件名为 configtx.yaml，文件内容如下所示：

Profiles:

```

TestOrgsOrdererGenesis:
  Orderer:
    <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
  Consortiums:
    SampleConsortium:
      Organizations:
        - *bc_org1
        - *bc_org2
        - *bc_org3

TestOrgsChannel:
  Consortium: SampleConsortium
  Application:
    <<: *ApplicationDefaults
    Organizations:

```

- \*bc\_org1
- \*bc\_org2
- \*bc\_org3

TestOrgsChannelbyone:

Consortium: SampleConsortium

Application:

<<: \*ApplicationDefaults

Organizations:

- \*bc\_org1
- \*bc\_org2
- \*bc\_org3

TestOrgsChannelbytwo:

Consortium: SampleConsortium

Application:

<<: \*ApplicationDefaults

Organizations:

- \*bc\_org1
- \*bc\_org3

TestOrgsChannelbythree:

Consortium: SampleConsortium

Application:

<<: \*ApplicationDefaults

Organizations:

- \*bc\_org2
- \*bc\_org3

Organizations:

- &OrdererOrg

Name: OrdererOrg

ID: OrdererMSP

MSPDir: /var/qklszzn/crypto-config/ordererOrganizations/qklszzn.com/msp

- &bc\_org1

Name: BcOrg1MSP

ID: BcOrg1MSP

MSPDir: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/msp

AnchorPeers:

- Host: peer0.org1.qklszzn.com
- Port: 7051

- &bc\_org2

Name: BcOrg2MSP

ID: BcOrg2MSP

MSPDir: /var/qklszzn/crypto-config/peerOrganizations/org2.qklszzn.com/msp

AnchorPeers:

- Host: peer0.org2.qklszzn.com

```

        Port: 7051
    - &bc_org3
        Name: BcOrg3MSP
        ID: BcOrg3MSP
        MSPDir: /var/qklszzn/crypto-config/peerOrganizations/org3.qklszzn.com/msp
        AnchorPeers:
            - Host: peer0.org3.qklszzn.com
              Port: 7051

Orderer: &OrdererDefaults

OrdererType: kafka

Addresses:
    - orderer.qklszzn.com:7050

BatchTimeout: 2s

BatchSize:

    MaxMessageCount: 10
    AbsoluteMaxBytes: 98 MB
    PreferredMaxBytes: 512 KB

Kafka:
    Brokers:
        - 127.0.0.1:9092
    Organizations:

Application: &ApplicationDefaults
    Organizations:

```



**注意** 上述配置文件中请注意 Kafka 服务器的部署地址。

configtx.yaml 文件中的 Profiles 节点的子节点 TestOrgsOrdererGenesis 为系统创始块的相关配置信息。通过 configtxgen 模块可以生成创始块的初始化文件。生成的命令如下所示：

```
configtxgen -profile TestOrgsOrdererGenesis -outputBlock
./orderer.genesis.block
```

-profile TestOrgsOrdererGenesis 就是在 configtx.yaml 配置文件 Profiles 节点的子节点名称。

configtxgen 命令执行完成之后会在根目录下面生成创始块文件 orderer.genesis.block。



系统的创始块生成之后,可以创建 Orderer 模块的配置文件来启动 Orderer 模块, Orderer 模块的配置文件名为 orderer.yaml。由于配置文件的内容太多,这里以 Fabric 源代码中示例配置文件为基础进行修改。示例文件夹的路径为:

<https://github.com/hyperledger/fabric/blob/release/sampleconfig/orderer.yaml>

首先创建一个文件夹用来存放 Orderer 模块相关的配置文件和数据文件。创建文件夹的命令如下:

```
mkdir -p /var/qklszzn/order
```

需要修改的内容如下所示:

General:

TLS:

```
PrivateKey: /var/qklszzn/crypto-config/ordererOrganizations/qklszzn.com/
orderers/orderer.qklszzn.com/tls/server.key
Certificate: /var/qklszzn/crypto-config/ordererOrganizations/qklszzn.com/
orderers/orderer.qklszzn.com/tls/server.crt
```

RootCAs:

```
- /var/qklszzn/crypto-config/ordererOrganizations/qklszzn.com/orderers/
orderer.qklszzn.com/tls/ca.crt
```

GenesisMethod: file

GenesisProfile: TestOrgsOrdererGenesis

GenesisFile: /var/qklszzn/orderer.genesis.block

```
LocalMSPDir: /var/qklszzn/crypto-config/ordererOrganizations/qklszzn.com/orderers/
orderer.qklszzn.com/msp
```

LocalMSPID: OrdererMSP

FileLedger:

Location: /var/qklszzn/order/production/orderer

Prefix: hyperledger-fabric-ordererledger

Kafka:

Retry:

ShortInterval: 1s

ShortTotal: 30s

Verbose: true

将修改好的 Orderer 配置文件 orderer.yaml 保存到文件夹 /var/qklszzn/order 中,现在可以启动 Orderer 模块了。Orderer 模块的启动方式有两种,分别是直接启动和通过 Docker 容器的方式进行启动,本章所有内容一律采用直接启动的方式。在 orderer.yaml 文件所在的目录中执行以下命令启动 Orderer 模块。



注意 在启动之前有一点非常重要,在配置文件中我们用域名替代了 IP 地址,本例中的域名是测试域名,需要在服务器中进行域名的映射才能访问。在编写本书的时候,本



例运行服务器的地址为 192.168.23.212, 操作系统为 Ubuntu。采用的域名映射命令如下:

```
vi /etc/hosts
```

在打开文件中输入以下内容:

```
192.168.23.212 orderer.qklszzn.com
192.168.23.212 peer0.org1.qklszzn.com
```

保存内容

Orderer 模块的启动命令如下:

```
orderer start
```

如果想让 Orderer 模块常驻后台运行, 采用以下命令:

```
orderer start >> log_orderer.log 2>&1 &
```

## 2. 对后续操作产生影响的地址和配置文件

在 configtxgen 模块的配置文件 configtx.yaml 中有一些配置信息非常重要。现列举如下:

### (1) Orderer 相关的配置信息

Orderer 相关的配置信息内容如下:

```
- &OrdererOrg

    Name: OrdererOrg
    ID: OrdererMSP
    MSPDir: /var/qklszzn/crypto-config/ordererOrganizations/qklszzn.com/msp
```

需要注意的有这样节点:

- **ID 属性:** ID 属性的值表示该 Orderer 节点的名字, 在后面的 Orderer 模块的配置文件 orderer.yaml 中, 属性 LocalMSPID 的值必须同此处的值一致, 否则系统会提示错误。
- **MSPDir 属性:** MSPDir 属性表示当前 Orderer 节点的账号文件的路径 (关于 Fabric 的账号情况参考本书的第 6 章), 必须指向由 cryptogen 工具生成的相关目录。注意: 如果采用集群的方式在其他机器上布置 Orderer 节点, 那么其他机器上的 Orderer 节点中配置文件的 MSP 文件夹的路径和 MSPDir 属性的值必须和这里配置是一致的, 否则 Orderer 无法启动。

### (2) 组织相关配置信息

本例中有三个组织 bc\_org1、bc\_org2、bc\_org3。他们的配置信息都是一样的。我们以组织 bc\_org1 的配置为例来说明 configtx.yaml 配置文件中组织相关配置信息在后续操作中的

重要作用。

```
- &bc_org1
  Name: BcOrg1MSP
  ID: BcOrg1MSP
  MSPDir: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/msp
  AnchorPeers:
    - Host: peer0.org1.qklszzn.com
      Port: 7051
```

上述配置信息有以下需要注意的地方：

- **ID 属性**：ID 属性的值是当前组织在整个系统中唯一的标识符，在后面的很多操作中都需要用到该值。注意：任何两个组织的 ID 属性都不能相同，否则系统无法运行。
- **MSPDir 属性**：MSPDir 属性的值存放的是当前组织的账号（关于 Fabric 的账号情况参考本书第 6 章的内容）。组织中的任何节点，不管运行在哪个服务器上，都必须保证把账号文件存储在和 MSPDir 属性的值设置的路径。在多机部署，特别是非 Docker 部署的时候，这一点尤其重要。在作者运营的区块链技术论坛中，很多问题都和这个路径有关系。读者在使用 Fabric 的过程中，如果遇到错误，首先请检查路径是否正确。
- **AnchorPeers 属性**：AnchorPeers 属性主要是配置了锚节点的信息。锚节点是一个 Peer 服务器节点，该节点专门负责组织和其他组织的信息交互。锚节点非常重要，如果锚节点配置错误，那么组织将无法被其他组织感知到。在加入一个组织的时候需要验证新组织的锚节点的服务器地址可以被访问到。同时新组织也需要验证已有组织的锚节点是否可以被正确访问。

### 10.2.3 启动第一个 Peer

#### 1. 设置 Peer 模块的配置文件

Peer 模块的配置文件的内容比较多，我们选择在 Fabric 源码中提供的模板文件进行修改。模板文件的路径如下：

```
https://github.com/hyperledger/fabric/blob/release/sampleconfig/core.yaml
```

由于 core.yaml 中的配置信息太多，这里只列出需要修改的部分。在修改配置文件之前，首先创建一个文件夹用来存放 Peer 模块的配置文件和数据文件。创建文件夹的命令如下所示：

```
mkdir -p /var/qklszzn/peer
```

Peer 模块配置文件需要修改的部分如下：

```

peer:

  id: peer0.org1.qklszzn.com
  networkId: dev
  listenAddress: 0.0.0.0:7051
  address: peer0.org1.qklszzn.com:7051
  addressAutoDetect: false
  gomaxprocs: -1

  gossip:
    bootstrap: peer0.org1.qklszzn.com:7051
    useLeaderElection: true
    orgLeader: false
    externalEndpoint: peer0.org1.qklszzn.com:7051
  events:

    address: 0.0.0.0:7053
  tls:
    enabled: false
    cert:
      file: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer0.org1.qklszzn.com/tls/server.crt
    key:
      file: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer0.org1.qklszzn.com/tls/server.key
    rootcert:
      file: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer0.org1.qklszzn.com/tls/ca.crt
      filePath: /var/qklszzn/peer/production
      mspConfigPath: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer0.org1.qklszzn.com/msp
      localMspId: BcOrg1MSP

```

将修改好的 Peer 模块配置文件 `core.yaml` 保存到文件夹 `/var/qklszzn/peer` 中。现在可以启动 Peer 模块了。Peer 模块的启动方式有两种，分别是直接启动和通过 Docker 容器的方式进行启动，本章所有内容一律采用直接启动的方式。在 `core.yaml` 文件所在的目录中执行以下命令启动 Orderer 模块。

```

export set FABRIC_CFG_PATH=/var/qklszzn/peer
peer node start

```

如果想让 Peer 模块常驻后台运行，采用以下命令：

```

export set FABRIC_CFG_PATH=/var/qklszzn/peer
peer node start >> log_peer.log 2>&1 &

```

## 2. 对后续操作产生影响的地址和配置信息

Peer 模块的配置文件 `core.yaml` 有一些属性比较重要，如果处理不当可能会引起错误。



具体需要关注以下配置属性。

- **bootstrap** : bootstrap 表示组织内部的一个 Peer 服务器节点的地址, 当 Peer 模块启动之后首先需要向该节点服务器发送请求同步本组织的相关信息。如果是第一个 Peer 节点可以填写本机的地址, 如果组织中已经存在 Peer 节点, 那么可以添加任何一个已经存在的 Peer 节点的地址。
- **listenAddress**: 本机监听的地址。
- **fileSystemPath**: Peer 节点本地数据文件的存储路径。
- **mspConfigPath** : Peer 节点账号文件的目录路径。这些文件是通过 cryptogen 模块生成的。
- **localMspId**: Peer 节点所属组织的编号, 该编号在 cryptogen 模块的配置文件中设定, 设定后一般不需要修改。

#### 10.2.4 Channel 的创建和加入

Peer 节点启动后还不能进行任何工作, 首先需要创建 Channel, 创建成功之后需要将当前的 Peer 加入到新创建的 Channel 中。创建 Channel 需要使用 configtxgen 模块, 并在其配置文件中配置 Channel 的信息。在 10.2.2 节中已经配置好了相关 Channel 的配置信息, 我们截取其中的一段来说明配置信息中包含哪些要素, 如果读者想了解详细的配置信息, 可参考 10.2.2 节中的配置文件。

下面是一个名为 TestOrgsChannel 的 Channel 的配置信息。

```
TestOrgsChannel:
  Consortium: SampleConsortium
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *bc_org1
      - *bc_org2
      - *bc_org3
```

该 Channel 中包含三个组织, bc\_org1、bc\_org2、bc\_org3。通道生成之后只有这三个组织的内 Peer 节点才能够加入到 Channel 中或者访问 Channel 中的交易信息。通过下面的步骤可以生成 Channel 的创始块文件, 并将已经启动的 Peer 加入到新创建的 Channel 中。

##### 第一步: 创建 Channel 提案文件

创建 Channel 提案文件的命令如下所示:

```
cd /var/qklszzn
configtxgen -profile TestOrgsChannel -outputCreateChannelTx
./fabricchannel.tx -channelID fabricchannel
```

上述命令执行完成后，在当前目录中会生成名为 `fabricchannel.tx` 的文件。在生成 Channel 的命令中有一些参数是需要注意的，譬如：

- **-profile**。-profile 参数设置的值必须和 10.2.2 节中定义的配置文件 `configtx.yaml` 中的 Profiles 节点中定义的子节点完全一致，否则命令会出错。

### 第二步：创建锚节点通知提案

锚节点通知提案负责将 Channel 创建的消息通知告诉锚节点，锚节点负责通知其他组织 Channel 创建的消息。创建锚节点通知提案的命令如下所示：

```
configtxgen -profile TestOrgsChannel -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID fabricchannel -asOrg BcOrg1MSP
```

上述命令执行完成之后在当前目录生成名为 `Org1MSPanchors.tx` 的文件。

### 第三步：创建 Channel 创始块

创建 Channel 创始块命令如下所示：

```
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
export set CORE_PEER_MSPCONFIGPATH=/var/qklszn/crypto-config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp

peer channel create -t 50 -o orderer.qklszn.com:7050 -c fabricchannel -f /var/qklszn/fabricchannel.tx
```

- **CORE\_PEER\_LOCALMSPID**。环境变量 `CORE_PEER_LOCALMSPID` 表示创建 Channel 的组织的编号。
- **CORE\_PEER\_MSPCONFIGPATH**。环境变量 `CORE_PEER_MSPCONFIGPATH` 表示执行创建 Channel 的用户账号。

上述命令执行成功之后会在当前文件夹中生成名为 `fabricchannel.block` 的文件。

### 第四步：加入 Channel

现在可以将前面启动的 Peer 节点加入到 Channel 中。加入 Channel 的命令如下所示：

```
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszn/crypto-config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp

peer channel join -b /var/qklszn/fabricchannel.block
```

### 第五步：通知锚节点

通知锚节点的命令如下所示：

```
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszn/crypto-config/peerOrganizations/
```



```
org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

```
peer channel update -o orderer.qklszzn.com:7050 -c fabricchannel -f /var/qklszzn/
Org1MSPanchors.tx
```

通过上面 5 个步骤, 可以把 10.2.3 节中启动的 Peer 节点加入到名为 fabricchannel 的 Channel 中。

对后续操作产生影响的配置信息如下:

在创建并加入 Channel 的过程中会产生三个文件: fabricchannel.tx、fabricchannel.block、Org1MSPanchors.tx。三个文件中 fabricchannel.block 是通道的创始块文件, 其余的 Peer 节点如果想要加入名为 fabricchannel 的 Channel, 必须首先获取 fabricchannel 的 block 文件。在第一步(创建 Channel 创始块)中, 创建 Channel 创始块的命令中有个选项 -channelID, 该选项后面的值为 Channel 的名字, 一旦创建无法更改。本章后续内容中所有需要使用 Channel 名称的地方, 一律使用 fabricchannel。

### 10.2.5 启动当前组织的 Fabric-ca

在项目初始化的过程中, 采用了 cryptogen 模块生成了相关的账号, 这种方式无法满足动态添加用户账号信息的场景, 因此需要将 Fabric-ca-server 绑定到当前组织中。Fabric-ca-server 的安装请参考 6.3.1 节的相关内容。在 Fabric-ca-server 的配置文件 fabric-ca-server-config.yaml 中设置以下信息完成绑定过程。

```
ca:
  name: ca-org1
  keyfile: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/ca/
ad4cb5ecd7db0f2442c76ac9c09d8d82896a8b6cfe1e0fbc077d0e28d6b1f877_sk
  certfile: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/ca/ca.
org1.qklszzn.com-cert.pem
```



**注意** keyfile 属性的值会因为环境的差异而不同。

绑定信息设置完成后, 可以启动 Fabric-ca-server 服务器。

### 10.2.6 测试 Chaincode 的部署和开发

我们以 Fabric 源代码中的示例 Chaincode 为例, 部署测试 Chaincode。首先需要设置相关环境变量, 设置相关环境变量的命令如下所示:

```
export set FABRIC_CFG_PATH=/var/qklszzn/peer
export set ORDERER_GENERAL_LOGLEVEL=debug
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
```



```
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/crypto-config/peerOrganizations/
org1.qklszzn.com/users/Admin@org2.qklszzn.com/msp
```

### (1) 安装 Chaincode

```
peer chaincode install -n cc_qklszzcc -v 1.0 -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02
```

### (2) 实例化 Chaincode

```
peer chaincode instantiate -o orderer.qklszzn.com:7050 -C fabricchannel -n cc_
qklszzcc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('BcOrg1MSP.member',
'BcOrg2MSP.member')"
```

### (3) 调用 Chaincode 的 invoke 方法

```
peer chaincode invoke -o 192.168.23.212:7050 -C fabricchannel -n cc_qklszzcc
-c '{"Args":["invoke","a","b","1"]}'
```

### (4) 调用 Chaincode 的 query 方法

```
peer chaincode query -C fabricchannel -n cc_qklszzcc -c '{"Args":["query","a"]}'
```

上述命令都能正确执行的话说明前面的设置都是正确的。本例中 Chaincode 采用的 Fabric 是源码自带的示例 Chaincode，如果想要自己编写示例 Chaincode，请参考第 7 章。

## 10.2.7 客户端的开发

Chaincode 编写完成之后可以在客户端通过命令调用，在实际项目中这显然是不够的，需要通过 Fabric 提供的 Grpc 接口在客户端程序中调用 Fabric。第 8 章中详细介绍了如何通过 Fabric 的 SDK 调用 Fabric，这里不再复述。这里我们给出用 Nodejs 调用本例中 Chaincode 的示例。本书在编写测试用例的时候将 Fabric 和 Fabric-ca-server 部署的服务器为 192.168.23.212，下面的示例代码中将会采用该服务器的地址。读者如果想自己测试，请将相关的 IP 地址修改成自己测试服务器的地址或者域名。Node.js 调用本例中 Chaincode 的示例代码如下所示：

```
var co = require('co');
var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
```





```

var FabricCAService = require('fabric-ca-client');

var log4js = require('log4js');
var logger = log4js.getLogger('Helper');
logger.setLevel('DEBUG');

var tempdir = "/temp/fabric-client-kvs";

var client = new hfc();
// 1、设置相关的环境变量

// 设置用于存储相关文件路径
var cryptoSuite = hfc.newCryptoSuite()
cryptoSuite.setCryptoKeyStore( hfc.newCryptoKeyStore({ path:tempdir } ) )
client.setCryptoSuite(cryptoSuite)

// 创建 CA 客户端
var caClient = new FabricCAService('http://192.168.23.212:7054',null, '', crypto-
Suite);

// 创建账本
var channel = client.newChannel('fabricchannel');
// 创建 order
var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);
// 创建节点

var peer = client.newPeer('grpc://192.168.23.212:7051');
channel.addPeer(peer);

co(( function *() {
    let member = yield getOrgAdmin4Local();
    let resultpeerinfo = yield channel.queryInfo(peer)
    console.info( JSON.stringify(resultpeerinfo) )
})
)())

// 通过 cryptogen 生成的账号调用 Fabric
function getOrgAdmin4Local() {

    var keyPath = "/var/qklszn/crypto-config/peerOrganizations/org1.qklszn.
com/users/Admin@org1.qklszn.com/msp/keystore";
    var keyPEM = Buffer.from(readAllFiles(keyPath) [0]).toString();
    var certPath = "/var/qklszn/crypto-config/peerOrganizations/org1.qklszn.
com/users/Admin@org1.qklszn.com/msp/signcerts";

```



```

var certPEM = readAllFiles(certPath)[0].toString();

return hfc.newDefaultKeyValueStore({
    path: tempdir
}).then((store) => {
    client.setStateStore(store);

    return client.createUser({
        username: 'Admin',
        mspid: 'Org1MSP',
        cryptoContent: {
            privateKeyPEM: keyPEM,
            signedCertPEM: certPEM
        }
    });
});
});

};

function readAllFiles(dir) {
    var files = fs.readdirSync(dir);
    var certs = [];
    files.forEach((file_name) => {
        let file_path = path.join(dir, file_name);
        let data = fs.readFileSync(file_path);
        certs.push(data);
    });
    return certs;
}

```

### 10.2.8 启动本组织的其他 Peer

在前面的操作中我们已经启动了 Orderer 节点，然后启动了组织 BcOrg1MSP 的一个节点。现在需要启动组织 BcOrg1MSP 的第二个节点，并且将这个节点加入到之前创建的名为 fabricchannel 的 Channel 中。为了达到更好的演示效果，我们建议在另外独立的服务器上面启动组织 BcOrg1MSP 的第二个 Peer 节点。

#### 1. 启动 Peer 节点

在第二台服务器上我们需要创建和一个第一台服务器同样的路径，并且将相关的配置文件复制到同等目录中。

创建目录的命令如下所示:

```
mkdir /var/qklszzn/crypto-config
```

将 10.2.1 节中创建的证书文件部分复制到目录 /var/qklszzn/crypto-config 中, 成功复制后可以通过以下命令的结果核对复制是否正确:

```
tree -L 2
```

```
.
├── peerOrganizations
│   └── org1.qklszzn.com
│       ├── ca
│       │   ├── ad4cb5ecd7db0f2442c76ac9c09d8d82896a8b6cfe1e0fbc077d0e28d6b1f877_sk
│       │   └── ca.org1.qklszzn.com-cert.pem
│       ├── msp
│       │   ├── admincerts
│       │   ├── cacerts
│       │   └── tlscacerts
│       ├── peers
│       │   └── peer1.org1.qklszzn.com
│       ├── tlsca
│       │   ├── 6586181932b977b05553539523d5635940c3cbcbe9ae512331c6103d6de687d3_sk
│       │   └── tlsca.org1.qklszzn.com-cert.pem
│       └── users
│           └── Admin@org1.qklszzn.com
```

上诉命令成执行后, 我们可以发现相关证书文件的数量和第一个 Peer 节点的服务器不一样。因为在运行组织 BcOrg1MSP 的第二个 Peer 节点的服务器上, 我们需要运行一个 Peer 节点, 而不需要运行 Orderer, 所以我们只将需要的证书复制过来。这里我们使用了 peer1.org1.qklszzn.com 的账号来生成组织 BcOrg1MSP 的第二个 Peer 节点。第二个 Peer 节点的配置文件还是以 Fabric 源码中提供的 Peer 模块的配置文件模板为基础进行修改。由于本书篇幅的限制, 这里我们仅列出修改过的内容, 具体内容如下所示:

```
peer:
  id: peer1.org1.qklszzn.com
  networkId: dev
  listenAddress: 0.0.0.0:7051
  address: peer1.org1.qklszzn.com:7051
  addressAutoDetect: false
  gomaxprocs: -1
  gossip:
    bootstrap: peer1.org1.qklszzn.com:7051
    useLeaderElection: true
    orgLeader: false
    externalEndpoint: peer1.org1.qklszzn.com:7051
  events:
    address: 0.0.0.0:7053
```

```

tls:
  enabled: false
  cert:
    file: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer1.org1.qklszzn.com/tls/server.crt
  key:
    file: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer1.org1.qklszzn.com/tls/server.key
  rootcert:
    file: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer1.org1.qklszzn.com/tls/ca.crt

  filePath: /var/qklszzn/production
  mspConfigPath: /var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
peers/peer1.org1.qklszzn.com/msp
  localMspId: BcOrg1MSP

```

配置文件修改完成后便可以启动组织 BcOrg1MSP 的第二个 Peer 节点。启动命令如下所示：

```

export set FABRIC_CFG_PATH=/var/qklszzn/peer
peer node start

```

如果想让 Peer 模块常驻后台运行，采用以下命令：

```

export set FABRIC_CFG_PATH=/var/qklszzn/peer
peer node start >> log_peer.log 2>&1 &

```

## 2. 将 Peer 节点加入到 Channel 中

组织 BcOrg1MSP 的第二个 Peer 节点成功启动后，需要加入到名为 fabricchannel 的 Channel 中以便获取账本信息。在加入之前首先需要获取通道的初始块文件，获取初始块的方式有两种：

### (1) 直接使用现存通道的初始块文件

在 10.2.4 节的第三步正确执行完成后会生成一个名为 fabricchannel.block，将该文件直接复制过来即可。

### (2) 通过命令行工具导出

如果之前生成的初始块丢失了，可以通过命令工具从现有通道中导出初始块文件，导出 Channel 初始块的命令如下所示：

```

export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER MSPCONFIGPATH=/var/qklszzn/peer/crypto-config/peerOrganizations/
org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer channel fetch 0 fabricchannel.block -c fabricchannel -o orderer.qklszzn.
com:7050

```



上述命令中需要在已经加入 Channel 的 Peer 节点的环境中运行。Channel 的初始块创建完成之后执下面的命令:

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer1.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/peer/crypto-config/peerOrganizations/
org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

```
peer channel join -b /var/qklszzn/fabricchannel.block
```

上述命令成功执行后, 组织 BcOrg1MSP 的第二个 Peer 节点已经成功加入了名为 fabric-channel 的 Channel 中。

### 10.2.9 其他组织 Peer 节点的加入

在前面的操作中, 我们已经成功启动了组织 BcOrg1MSP 的两个 Peer 节点, 现在我们要启动组织 BcOrg2MSP 的 Peer 节点了。启动顺序和 10.2.8 节是一样的, 需要注意的是组织 BcOrg2MSP 的 Peer 节点所使用的证书的路径。具体的步骤如下:

#### 1. 启动组织 BcOrg2MSP 的 peer 节点

组织 BcOrg1MSP 的第二个 Peer 节点的配置文件:

```
mkdir /var/qklszzn/crypto-config
```

将 10.2.1 节中创建的证书文件部分复制到目录 /var/qklszzn/crypto-config 中, 具体复制的文件可以参考以下命令的执行结果:

```
tree -L 4
```

```
.
├── peerOrganizations
│   └── org2.qklszzn.com
│       ├── ca
│       │   ├── 5445f00612f16cbf86f1e34a0013d22568379f397729605505bc66c323d7ac48_sk
│       │   └── ca.org2.qklszzn.com-cert.pem
│       ├── msp
│       │   ├── admincerts
│       │   ├── cacerts
│       │   └── tlscacerts
│       ├── peers
│       │   ├── peer0.org2.qklszzn.com
│       │   ├── peer1.org2.qklszzn.com
│       │   └── peer2.org2.qklszzn.com
│       ├── tlsca
│       │   ├── c2f5807a9e91b8cb76ccee57290f5288e5c142449e762c415354a6dd0697dd27_sk
│       │   └── tlsca.org2.qklszzn.com-cert.pem
│       └── users
```

```

|—— Admin@org2.qklszzn.com
|—— User1@org2.qklszzn.com
|—— User2@org2.qklszzn.com
|—— User3@org2.qklszzn.com
|—— User4@org2.qklszzn.com

```



注意 上述命令使用了目录 `org2.qklszzn.com` 下面的证书文件。

账号证书文件准备好之后，开始复制设置 Peer 节点的配置文件 `core.yaml`。我们还是和 10.2.3 节介绍的一样，在 Fabric 源码中提供的样例配置文件的基础上进行修改，需要修改的内容如下：

```

peer:
  id: peer0.org1.qklszzn.com
  networkId: dev
  listenAddress: 0.0.0.0:7051
  address: peer0.org2.qklszzn.com:7051
  addressAutoDetect: false
  gomaxprocs: -1
  gossip:
    bootstrap: peer0.org2.qklszzn.com:7051
    useLeaderElection: true
    orgLeader: false
    externalEndpoint: peer0.org2.qklszzn.com:7051
  events:
    address: 0.0.0.0:7053
  tls:
    enabled: false
    cert:
      file: /var/qklszzn/crypto-config/peerOrganizations/org2.qklszzn.com/
peers/peer0.org2.qklszzn.com/tls/server.crt
    key:
      file: /var/qklszzn/crypto-config/peerOrganizations/org2.qklszzn.com/
peers/peer0.org2.qklszzn.com/tls/server.key
    rootcert:
      file: /var/qklszzn/crypto-config/peerOrganizations/org2.qklszzn.com/
peers/peer0.org2.qklszzn.com/tls/ca.crt

  filePath: /var/qklszzn/peer/production
  mspConfigPath: /var/qklszzn/crypto-config/peerOrganizations/org2.qklszzn.com/
peers/peer0.org2.qklszzn.com/msp
  localMspId: BcOrg2MSP

```

执行以下命令启动 Peer 节点：

```

export set FABRIC_CFG_PATH=/var/qklszzn/peer
peer node start

```

如果想让 Peer 模块常驻后台运行, 采用以下命令:

```
export set FABRIC_CFG_PATH=/var/qklszzn/peer
peer node start >> log_peer.log 2>&1 &
```

本例中 peer0.org2.qklszzn.com 节点的部署地址的 IP 为 192.168.23.213, 在后面 SDK 部分需要用到这个 IP 地址, 读者如果需要演示本示例请根据自己的实际环境予以调整。

## 2. 将组织 BcOrg2MSP 的 Peer 节点加入到 Channel 中

直接使用现存通道的初始块文件

复制 10.2.4 节的第三步正确执行完成之后生成的初始块文件 fabricchannel.block, 将该文件复制到当前服务器中。Channel 的初始块创建完成后执行下面的命令:

```
export set CORE_PEER_LOCALMSPID=BcOrg2MSP
export set CORE_PEER_ADDRESS=peer0.org2.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/peer/crypto-config/peerOrganizations/
org2.qklszzn.com/users/Admin@org2.qklszzn.com/msp
peer channel join -b /var/qklszzn/fabricchannel.block
```

上述命令成功执行后, 组织 BcOrg1MSP 的第二个 Peer 节点已经成功加入了名为 fabric-channel 的 Channel 中。在上述代码的执行中如果出现错误, 请做以下检查:

- **本地域名映射检查:** 由于还要和组织 BcOrg1MSP 的锚节点通信, 请检查是否已经在本机和组织 BcOrg1MSP 的锚节点做了映射。具体可以检查本机的 /etc/hosts 文件。以下是本例的参考:

```
192.168.23.212 peer0.org1.qklszzn.com
```

- **检查组织的配置文件:** 检查本地组织 BcOrg2MSP 组织的配置文件是否和 11.2.1 节中 cryptogen 模块的配置文件的配置路径完全一致。

### 10.2.10 背书交易的测试

到目前为止系统中已经有两个组织了, 这时需要测试如何通过客户端发起一笔需要两个组织共同完成背书的交易了。首先需用重新部署一个 Chaincode, 这个 Chaincode 的交易验证规则和以前的不一样, 需要经过组织 BcOrg1MSP 和组织 BcOrg2MSP 同时验证后才能生效。部署过程如下:

在组织 BcOrg1MSP 的 peer0.org1.qklszzn.com 上面部署 Chaincode:

```
export set FABRIC_CFG_PATH=/var/qklszzn/peer
export set ORDERER_GENERAL_LOGLEVEL=debug
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/fabricconfig/crypto-config/peer-
```

```
Organizations/org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp
```

```
peer chaincode install -n cc_endfinlshed -v 1.0 -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02
```

```
peer chaincode instantiate -o orderer.qklszzn.com:7050 -C fabricchannel -n cc_
endfinlshed -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "AND ('BcOrg1MSP.
member','BcOrg2MSP.member')"
```

```
peer chaincode query -C fabricchannel -n cc_endfinlshed -c '{"Args":["query","a"]}'
```

在组织 BcOrg2MSP 的 peer0.org2.qklszzn.com 上面部署 Chaincode:

```
export set FABRIC_CFG_PATH=/var/qklszzn/peer
export set ORDERER_GENERAL_LOGLEVEL=debug
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
export set CORE_PEER_ADDRESS=peer0.org2.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/fabricconfig/crypto-config/peer-
Organizations/org2.qklszzn.com/users/Admin@org2.qklszzn.com/msp
```

```
peer chaincode install -n cc_endfinlshed -v 1.0 -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02
```

```
peer chaincode query -C fabricchannel -n cc_endfinlshed -c '{"Args":["query",
"a"]}'
```

客户端代码示例如下:

```
var co = require('co');

var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
var FabricCAService = require('fabric-ca-client');

var hfc = require('fabric-client');

var log4js = require('log4js');
var logger = log4js.getLogger('Helper');
logger.setLevel('DEBUG');

var tempdir = "/temp/fabric-client-kvs";

var client
var caClient
```



```
var client = new hfc();

// 创建账本
var channel = client.newChannel('fabricchannel');

// 创建 order
var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);
// 创建节点

// 创建节点
var peer212 = client.newPeer('grpc://172.16.10.212:7051');
channel.addPeer(peer212);

var peer213 = client.newPeer('grpc://192.168.23.213:7051');
channel.addPeer(peer213);

co(( function *() {

    // 根据本地证书而不是依赖 CA 的方式获取管理员账号信息
    let member = yield getOrgUser4Local();

    // // / ===== 多方背书 =====

    let tx_id = client.newTransactionID();
    var request = {

        chaincodeId: "cc_endfinlshed",
        fcn: "invoke",
        args: ["a","b","1"],
        chainId: "fabricchannel",
        txId: tx_id
    };

    let chaincodeinvokresult = yield channel.sendTransactionProposal(request);

    var proposalResponses = chaincodeinvokresult[0];
    var proposal = chaincodeinvokresult[1];
    var header = chaincodeinvokresult[2];
    var all_good = true;

    for (var i in proposalResponses) {

        let one_good = false;
```

```

        if (proposalResponses && proposalResponses[0].response &&
            proposalResponses[0].response.status === 200) {
            one_good = true;
            console.info('transaction proposal was good');
        } else {
            console.error('transaction proposal was bad');
        }
        all_good = all_good & one_good;
    }

    if (all_good) {

        console.info(util.format(
            'Successfully : Status - %s, message - "%s", metadata - "%s", endorsement signature: %s',
            proposalResponses[0].response.status, proposalResponses[0].response.
message,
            proposalResponses[0].response.payload, proposalResponses[0].endor-
sement.signature));

        var request = {
            proposalResponses: proposalResponses,
            proposal: proposal,
            header: header
        };

        var transactionID = tx_id.getTransactionID();
        var sendPromise = yield channel.sendTransaction(request);

    }

    console.info(all_good)

}

)()

/**
 *
 * 根据 cryptogen 模块生成的账号通过 Fabric 接口进行相关的操作
 *
 */
function getOrgUser4Local() {

```

```

// 测试通过 CA 命令行生成的证书依旧可以成功发起交易
var keyPath = "/var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.com/
users/Admin@org1.qklszzn.com/msp/keystore";
var keyPEM = Buffer.from(readAllFiles(keyPath)[0]).toString();
var certPath = "/var/qklszzn/crypto-config/peerOrganizations/org1.qklszzn.
com/users/Admin@org1.qklszzn.com/msp/signcerts";
var certPEM = readAllFiles(certPath)[0].toString();

return hfc.newDefaultKeyValueStore({

    path:tempdir

}).then((store) => {
    client.setStateStore(store);

    return client.createUser({
        username: 'admin',
        mspid: 'adminpw',
        cryptoContent: {
            privateKeyPEM: keyPEM,
            signedCertPEM: certPEM
        }
    });
});
};

function readAllFiles(dir) {
    var files = fs.readdirSync(dir);
    var certs = [];
    files.forEach((file_name) => {
        let file_path = path.join(dir, file_name);
        let data = fs.readFileSync(file_path);
        certs.push(data);
    });
    return certs;
}

```

### 10.2.11 非初始化组织的加入

在 Fabric 项目运行的过程中, 有时候需要加入新的组织, 这时候需要对相关通道的创世块文件进行修改。本例中我们将演示如何在通道中加入非初始化组织。

#### 第一步: 生成新增加组织的配置文件

将以下内容保存到名为 crypto-org4.yaml 的配置文件中。

```

PeerOrgs:
  - Name: Org4

```

```
Domain: org4.qklszzn.com
CA:
  Country: China
  Province: shanghsi
  Locality: baoshan
Template:
  Count: 4
Users:
  Count: 3
```

执行以下命令生成新增组织的配置文件:

```
cryptogen generate --config=crypto-org4.yaml --output ./crypto-config
```

## 第二步: 启动 configtxlator

```
configtxlator start --hostname=0.0.0.0 --port=8188
```

## 第三步: 根据已经导出的创始块文件, 构造新的结构文件

导出需要修改的通道的新文件:

```
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qlksszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/crypto-config/peerOrganizations/
org1.qlksszn.com/users/Admin@org1.qlksszn.com/msp

peer channel fetch config fabricchannel.pb -c fabricchannel -o orderer.
qlksszn.com:7050
```

将导出的创始块文件转化成易于读写的 JSON 格式

```
curl -X POST --data-binary @fabricchannel.pb "http://127.0.0.1:8188/protolator/
decode/common.Block" > fabricchannel.json
```

## 第四步: 提取需要比较的地方

```
jq .data.data[0].payload.data.config fabricchannel.json > config.json
```

## 第五步: 构造增加组织的数据结构

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups": {"BcOrg4MSP":..
[1]}}}}}' config.json org4.json >& org4_config.json
```

org4.json 的格式如下:

```
{
  "groups": {},
  "mod_policy": "Admins",
  "policies": {
    "Admins": {
      "mod_policy": "Admins",
```



```

    "policy": {
      "type": 1,
      "value": {
        "identities": [
          {
            "principal": {
              "msp_identifier": "BcOrg4MSP",
              "role": "ADMIN"
            },
            "principal_classification": "ROLE"
          }
        ],
        "rule": {
          "n_out_of": {
            "n": 1,
            "rules": [
              {
                "signed_by": 0
              }
            ]
          }
        },
        "version": 0
      }
    },
    "version": "0"
  },
  "Readers": {
    "mod_policy": "Admins",
    "policy": {
      "type": 1,
      "value": {
        "identities": [
          {
            "principal": {
              "msp_identifier": "BcOrg4MSP",
              "role": "MEMBER"
            },
            "principal_classification": "ROLE"
          }
        ],
        "rule": {
          "n_out_of": {
            "n": 1,
            "rules": [
              {
                "signed_by": 0
              }
            ]
          }
        }
      }
    }
  },

```

```

        "version": 0
    },
    "version": "0"
},
"Writers": {
    "mod_policy": "Admins",
    "policy": {
        "type": 1,
        "value": {
            "identities": [
                {
                    "principal": {
                        "msp_identifier": "BcOrg4MSP",
                        "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                }
            ],
            "rule": {
                "n_out_of": {
                    "n": 1,
                    "rules": [
                        {
                            "signed_by": 0
                        }
                    ]
                }
            }
        }
    },
    "version": 0
},
"version": "0"
}
},
"values": {
    "MSP": {
        "mod_policy": "Admins",
        "value": {
            "config": {
                "admins": [
                    ""
                ],
                "crypto_config": {
                    "identity_identifier_hash_function": "SHA256",
                    "signature_hash_family": "SHA2"
                },
                "name": "BcOrg4MSP",
                "root_certs": [
                    ""
                ]
            },

```

```

        "tls_root_certs": [
            ""
        ]
    },
    "type": 0
},
"version": "0"
}
},
"version": "0"
}

```

上述模板文件还有 3 个属性需要绑定响应的证书文件, 分别是:

#### (1) admins

admins 属性中的内容对应的是组织 BcOrg4MSP 的 msp 文件中 admincerts 的文件内容, 将文件夹中的内容改为 Base64 格式的编码后, 复制到文件中。本例中对应文件的路径为 /var/qklszzn/crypto-config/peerOrganizations/org4.qklszzn.com/msp/admincerts。

#### (2) root\_certs

admins 属性中的内容对应的是组织 BcOrg4MSP 的 msp 文件中 cacerts 的文件内容, 将文件夹中的内容改为 Base64 格式的编码后, 复制到文件中。本例中对应文件的路径为 /var/qklszzn/crypto-config/peerOrganizations/org4.qklszzn.com/msp/cacerts。

#### (3) tls\_root\_certs

admins 属性中的内容对应的是组织 BcOrg4MSP 的 msp 文件中 tlscacerts 的文件内容, 将文件夹中的内容改为 Base64 格式的编码后, 复制到文件中。本例中对应文件的路径为 /var/qklszzn/crypto-config/peerOrganizations/org4.qklszzn.com/msp/tlscacerts。



**注意** 上面 3 个属性的值必须将对应文件采用 base64 格式编码之后才能保存到文件中。

### 第六步: 构建原来包含 3 个 org 的数据结构

```
curl -X POST --data-binary @config.json "http://127.0.0.1:8188/protolator/encode/common.Config" > config.pb
```

### 第七步: 构建新组织 BcBcOrg4MSP 的数据结构

```
curl -X POST --data-binary @org4_config.json "http://127.0.0.1:8188/protolator/encode/common.Config" > org4_config.pb
```

### 第八步: 计算两个数据结构的差异

```
curl -X POST -F original=@config.pb -F updated=@org4_config.pb http://127.0.0.1:8188/configxlator/compute/update-from-configs -F channel=fabricchannel > config_update.pb
```

### 第九步：解析差异化数据结构为 Json 格式

```
curl -X POST --data-binary @config_update.pb "http://127.0.0.1:8188/protolator/decode/common.ConfigUpdate" | jq . > config_update.json
```

### 第十步：构造更新消息的数据结构

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"fabricchannel", "type":2}}, "data":{"config_update":"'$(cat config_update.json)'"}}}' > config_update_in_envelope.json
```

### 第十一步：构造更新数据结构

```
curl -X POST --data-binary @config_update_in_envelope.json "http://127.0.0.1:8188/protolator/encode/common.Envelope" > config_update_in_envelope.pb
```

### 第十二步：当前通道的所有组织对交易数据结构进行签名

新组织的加入必须征得其他组织的同意，因此需要更新数据进行签名，才能提交更新请求。本例中需要修改的 Channel 中已经有了 BcOrg1MSP、BcOrg2MSP、BcOrg3MSP 三个组织，由于我们将采用 BcOrg1MSP 的账号提交证书，因此本例只需要 BcOrg2MSP、BcOrg3MSP 这两个组织的签名。

组织 BcOrg2MSP 的签名命令：

```
export set CORE_PEER_LOCALMSPID=BcOrg2MSP
export set CORE_PEER_MSPCONFIGPATH=/var/qklszn/crypto-config/peerOrganizations/org2.qklszn.com/users/Admin@org2.qklszn.com/msp
```

```
peer channel signconfigtx -o orderer.qklszn.com:7050 -f config_update_in_envelope.pb
```

组织 BcOrg3MSP 的签名命令：

```
export set CORE_PEER_LOCALMSPID=BcOrg3MSP
export set CORE_PEER_MSPCONFIGPATH=/var/qklszn/crypto-config/peerOrganizations/org3.qklszn.com/users/Admin@org3.qklszn.com/msp
```

```
peer channel signconfigtx -o orderer.qklszn.com:7050 -f config_update_in_envelope.pb
```

### 第十三步：执行更新交易

利用组织 BcOrg1MSP 的账号发起更新 Channel 的请求：

```
export set FABRIC_CFG_PATH=/var/qklszn
export set CORE_PEER_LOCALMSPID=Org1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszn/crypto-config/peerOrganizations/org1.qklszn.com/users/Admin@org1.qklszn.com/msp
```

```
peer channel update -o orderer.qklszn.com:7050 -c fabricchannel -f config_update_in_envelope.pb
```



#### 第十四步: 检查修改是否成功

导出名为 fabricchannel 的 Channel 配置块:

```
export set CORE_PEER_LOCALMSPID=BcOrg1MSP
export set CORE_PEER_ADDRESS=peer0.org1.qklszzn.com:7051
export set CORE_PEER_MSPCONFIGPATH=/var/qklszzn/crypto-config/peerOrganizations/
org1.qklszzn.com/users/Admin@org1.qklszzn.com/msp

peer channel fetch config fabricchanneladdorg4.block -c fabricchannel -o orderer.
qklszzn.com:7050
```

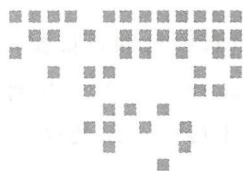
将创始块的内容转换成 JSON 的格式:

```
curl -X POST --data-binary @fabricchanneladdorg4.block http://127.0.0.1:8188/protolator/
decode/common.Block > ./fabricchanneladdorg4.json
```

打开文件 fabricchanneladdorg4.json, 检查里面的内容是否包含新添加的组织。如果发现组织 BcOrg4MSP 相关的信息, 则证明组织添加已经成功了。添加成功后的组织 BcOrg4MSP 便可以加入已经存在的系统中。

## 10.3 本章小结

本章内容是对前面章节内容的一次总结, 通过一个例子来介绍 Fabric 相关的模块如何运用到实际的项目中。通过阅读本章内容, 读者可以初步了解如何开发一个基于 Fabric 技术平台的项目。



## 基于 Fabric 的区块链浏览器项目实战

Fabric 浏览器可以用来查看 Fabric 系统的内部信息、比如区块数、区块详细信息、交易数目、交易详细信息。通过区块链浏览器项目，读者可以详细了解系统内置的 Chaincode 的使用，以及 Fabric 的内部结构。目前 Fabric 官方的浏览器项目 Blockchain-explorer (<https://github.com/hyperledger/blockchain-explorer>)，是由本书作者开发完成后，将全部代码捐献给 Hyperledger 项目组的，目前是 Hyperledger 官方唯一指定的浏览器。

本章将以该项目为蓝本给大家介绍如何通过 Fabric 系统提供的接口开发一个 Fabric 浏览器。Fabric 浏览器实战项目，我们将全部采用 Nodejs 作为开发语言，如果你对 Nodejs 不是非常了解可以先参考相关的资料。

### 11.1 项目介绍

Fabric 的 Peer 模块的提供了一些可以查询 Fabric 的系统信息的子命令，比如可以通过命令 `peer channel list` 查看当前 Peer 节点加入的通道。这些命令虽然可以获取 Fabric 的系统信息，但是操作不是非常方便，而且查询的结果显示不是那么友好。更重要的是这些命令必须在内网的主机上才能执行，如果用户在网络之外就无法访问到这些信息。在 Fabric 中提供了获取这些信息的 Grpc 接口，在 Hyperledger 项目组提供的 SDK 中，已经对这些接口进行了封装。因此基于这些 SDK 进行简单封装之后，将这些信息以 web 网站的形式发布出去，这样在任何地方都可以方便地获取 Fabric 的内部信息了。这就是 Fabric 浏览器的主要

功能。

Fabric 浏览器项目的目标是向读者演示如何获取 Fabric 的系统信息和对 Fabric 进行操作。为了提高开发效率,本例采用 Nodejs 开发。8.2.2 节中详细介绍了 Fabric Nodejs SDK 提供的操作接口,以及这些接口的调用方式,我们建议读者在阅读下面的内容前先熟悉 8.2.2 节的内容。在本例中我们采用了 Nodejs 的 express 框架作为 web 服务的引擎。

## 11.2 开发过程

在开发本实例之前请先按照第 4 章的内容正确完成里面的所有操作,在本实例运行前请保证 Peer 节点和 Orderer 节点处于运行状态。然后按照以下步骤完成本实例的开发工作。

### 11.2.1 项目准备

在本例中将使用 Nodejs 中的 express 框架作为 Web 框架,在项目开始之前需要通过 npm 工具安装 express 框架的相关包。安装前请先进入项目的目录,然后执行以下安装命令:

```
npm install express
```

执行完成后,express 模块的相关依赖包会被安装到项目文件中的 node\_modules 文件夹里。

### 11.2.2 项目开发

我们首先将 Fabric 相关的操作封装成一个类,方便后面调用。我们将这些接口代码保存在名为 fabricservice.js 的源代码文件中,具体的代码内容如下所示:

```
var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
var FabricCAService = require('fabric-ca-client');

var hfc = require('fabric-client');
var log4js = require('log4js');
```

```

var logger = log4js.getLogger('Helper');
logger.setLevel('DEBUG');

var tempdir = "/project/ws_nodejs/fabric_sdk_node_studynew/fabric-client-kvs";

let client = new hfc();
var channel = client.newChannel('roberttestchannel12');
var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);
var peer188 = client.newPeer('grpc://172.16.10.188:7051');
channel.addPeer(peer188);
var peer = client.newPeer('grpc://192.168.23.212:7051');
channel.addPeer(peer);

/**
 *
 * 获取 channel 的区块链信息
 * @returns {Promise.<TResult>}
 */
var getBlockchainInfo = function(){

    return getOrgUser4Local().then((user)=>{

        return channel.queryInfo(peer);

    }, (err)=>{

        console.log('error', e);

    })

}

/**
 * 根据区块链的编号获取区块的详细信息
 *
 * @param blocknum
 * @returns {Promise.<TResult>}
 */
var getblockInfobyNum = function (blocknum) {

    return getOrgUser4Local().then((user)=>{

        return channel.queryBlock(blocknum, peer, null);

    }, (err)=>{

```



```

        console.log('error', e);
    } )

}

/**
 * 根据区块链的哈希值获取区块的详细信息
 *
 * @param blocknum
 * @returns {Promise.<TResult>}
 */
var getblockInfoByHash = function ( blockHash ) {

    return getOrgUser4Local().then(( user )=>{

        return channel.queryBlockByHash(new Buffer(blockHash,"hex"),peer)

    } ,(err)=>{

        console.log('error', e);

    } )

}

/**
 *
 * 获取当前 Peer 节点加入的通道信息
 *
 * @param blocknum
 * @returns {Promise.<TResult>}
 */
var getPeerChannel = function ( ) {

    return getOrgUser4Local().then(( user )=>{

        return client.queryChannels(peer)

    } ,(err)=>{

        console.log('error', e);

    } )

}

/**

```

```

*
* 查询指定 peer 节点已经 install 的 chaincode
*
* @param blocknum
* @returns {Promise.<TResult>}
*
*/
var getPeerInstallCc = function ( ) {

    return getOrgUser4Local().then(( user )=>{

        return client.queryInstalledChaincodes(peer)

    }, (err)=>{

        console.log('error', e);

    } )

}

/**
*
* 查询指定 channel 中已经实例化的 Chaincode
*
* @param blocknum
* @returns {Promise.<TResult>}
*
*/
var getPeerInstantiatedCc = function ( ) {

    return getOrgUser4Local().then(( user )=>{

        return channel.queryInstantiatedChaincodes( peer )

    }, (err)=>{

        console.log('error', e);

    } )

}

/**
*
* 根据 cryptogen 模块生成的账号通过 Fabric 接口进行相关的操作
*

```

```

    * @returns {Promise.<TResult>}
    *
    */
function getOrgUser4Local() {

    // 测试通过 CA 命令行生成的证书依旧可以成功的发起交易
    var keyPath = "/project/fabric_resart/config_demo/org1/186/fabric-user/msp/
keystore";
    var keyPEM = Buffer.from(readAllFiles(keyPath)[0]).toString();
    var certPath = "/project/fabric_resart/config_demo/org1/186/fabric-user/msp//
signcerts";
    var certPEM = readAllFiles(certPath)[0].toString();

    return hfc.newDefaultKeyValueStore({

        path: tempdir

    }).then((store) => {
        client.setStateStore(store);

        return client.createUser({
            username: 'user87',
            mspid: 'Org1MSP',
            cryptoContent: {
                privateKeyPEM: keyPEM,
                signedCertPEM: certPEM
            }
        });
    });
};

function readAllFiles(dir) {
    var files = fs.readdirSync(dir);
    var certs = [];
    files.forEach((file_name) => {
        let file_path = path.join(dir, file_name);
        let data = fs.readFileSync(file_path);
        certs.push(data);
    });
    return certs;
}

exports.getBlockChainInfo = getBlockChainInfo;
exports.getBlockInfobyNum = getblockInfobyNum;
exports.getBlockInfobyHash = getblockInfobyHash;
exports.getPeerChannel = getPeerChannel;

```

```
exports.getPeerInstallCc = getPeerInstallCc;
exports.getPeerInstantiatedCc = getPeerInstantiatedCc;
exports.sendTransaction = sendTransaction;
```

将 web 服务保存在名为 fabricexplorer.js 文件，代码如下所示：

```
var co = require('co');
var fabricservice = require('./fabricservice.js')
var express = require('express');
```

```
var app = express();
```

// 获取当前通道的高度

```
app.get('/getchannelheight', function (req, res) {
```

```
  co( function * () {
```

```
    var blockchaininfo = yield fabricservice.getBlockChainInfo();
    res.send( blockchaininfo.height.toString() );
```

```
  }).catch((err) => {
    res.send(err);
  })
```

```
});
```

// 根据区块的编号获取区块的信息

```
app.get('/getblockInfobyNum', function (req, res) {
```

```
  co( function * () {
```

```
    var blockinfo = yield fabricservice.getblockInfobyNum(10);
    res.send( JSON.stringify( blockinfo ) );
```

```
  }).catch((err) => {
    res.send(err);
  })
```

```
});
```

// 根据区块的 Hash 值来获取区块的信息

```
app.get('/getblockInfobyHash', function (req, res) {
```

```
  co( function * () {
```

```
    var blockinfo = yield fabricservice.getblockInfobyHash("967400886bc2ec
a81168582211649be91fa8c0db905f301d214fefbd192000d2");
    res.send( JSON.stringify( blockinfo ) );
```



```

    }).catch((err) => {
        res.send(err);
    })
});

// 获取制定 Peer 节点加入的通道数
app.get('/getPeerChannel', function (req, res) {

    co( function * () {

        var peerJoinchannels = yield fabricService.getPeerChannel();
        res.send( JSON.stringify( peerJoinchannels  ) );

    })

});

app.get('/sendTransaction', function (req, res) {

    co( function * () {
        var blockinfo = yield fabricService.sendTransaction("cc_endfinlshed", "
invoke", ["a", "b", "1"], "roberttestchannel12");
        res.send( "success" );
    })

});

// 启动 http 服务
var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;

    console.log('Example app listening at http://%s:%s', host, port);
});

// 注册异常处理器
process.on('unhandledRejection', function (err) {
    console.error(err.stack);
});

process.on('uncaughtException', console.error);

```

现在可以启动这个简单的浏览器, 启动浏览器的命令如下所示:

```
node fabricexplorer.js
```

启动成功之后可以在浏览器中输入以下网址, 访问相关的内容。

```
// 获取当前 Perr 加入的通道  
http://localhost:3000/getPeerChannel
```

我们在上面提供了一个简单的例子，web 页面的样式和区块链关系不大，本书就不做详细讨论了。Fabric 的官方浏览器 Blockchain-explorer 是 Hyperledger 官方提供的浏览器，其界面非常精美，对界面比较感兴趣的读者可以参考一下。Blockchain-explorer 项目的地址如下所示：

```
https://github.com/hyperledger/blockchain-explorer
```

## 11.3 本章小结

本章通过一个简单的 Fabric 浏览器的例子向读者演示如何使用 Fabric 提供的系统接口获取 Fabric 的系统信息。通过本例读者可以初步了解 Fabric 系统接口的特性，对开发 Fabric 相关的应用有了初步的了解。

## 基于 Fabric 的供应链金融项目实战

供应链金融是近年来兴起的一类金融服务产品。在供应链金融产品中，供应商、核心企业、银行、金融机构等多方并存，共同参与交易的完成。由于参与方众多，其中涉及很多清算和结算的功能，这些功能采用传统的技术方案解决会产生很多中间环节，最终导致过程繁琐效率低下。而区块链的出现给供应链金融系统的实现提供了新的解决方案。本章将通过一个案例展示如何通过 Fabric 来解决供应链金融的一个典型问题。

### 12.1 供应链金融的背景知识和痛点

#### 12.1.1 供应链金融的背景知识

供应链金融本质上是一种金融服务。如果我们给供应链金融下一个定义的话，我们可以认为供应链金融是银行围绕核心企业，管理上下游中小企业的资金流和物流，并把单个企业的不可控风险转变为供应链企业整体的可控风险，通过获取各类信息，将风险控制在最低的金融服务。供应链金融不是一个独立的服务，而是在核心企业的领导下，由银行参与背书的，若干金融服务结构和相关核心企业按照一定的工作流程组合起来的金融服务组。供应链金融有以下常见服务场景：

- 应收账款融资
- 贸易数据融资
- 1+N 采购代理融资

- 票据融资
- 仓单融资
- 国际贸易

上述场景在供应链金融系统中不同的核心企业会涉及不同的组合。不管什么样的流程，这些服务都有一个共同的特点，那就是都需要引入核心企业、第三方企业（如物流公司）等新的风险控制变量，对供应链的不同节点提供封闭的授信及其他结算、理财等综合金融服务。其实质就是依靠风险控制变量，帮助企业盘活其流动资产从而解决融资问题。

### 12.1.2 供应链金融的痛点

在整个供应链金融系统中有一个非常关键的核心概念——“风控”，“风控”是需要通过及时获取真实的贸易背景信息来实现的。然而，在实际操作中我们发现，往往有很多的现实问题导致供应链系统的开展举步维艰。由于供应链运行过程中，各类信息分散保存在各个环节中，供应商的货物信息存储在供应商的仓储信息中，发货信息掌握在物流公司手里，资金信息分布在银行系统内，信息流信息则由核心企业掌握，整个供应链信息不透明、不流畅，各个参与主体难以了解交易事项的进展情况，不对称的信息影响了整个链条的效率，最终也导致整个供应链信用体系难以建立。而针对供应链贸易背景而提供的金融服务，也因为信息的不对称而难以顺利开展。

在供应链金融系统中，各个参与方（核心企业，第三方企业等）都有自己的相关系统，这样导致每一步的操作都会在各种异步的系统进行独立的确认操作。这些额外的中间环节的存在，导致这个过程非常繁琐，使得供应链金融设想的好处消失殆尽。相关机构往往会出于风控和成本的考虑而较为谨慎，有时甚至踟躇不前，很多原本可开展的供应链金融项目也因此被“误杀”。

针对这些痛点，整个业界都希望能从技术的角度解决这些问题。而区块链的出现给解决这些问题带来了曙光。

### 12.1.3 用 Fabric 解决供应链金融痛点的方法

Fabric 作为一个典型的区块链技术平台，在继承区块链技术去中心化、时序数据、集体维护、可编程的智能合约和安全可信这五大核心特点的基础上，增加了基于 PKI 的账号权限系统等适合企业应用的新特性。这些特性使得 Fabric 能够有效地解决供应链金融系统遇到的一些问题。比如 Fabric 区块链技术自带的时间戳与数据不可篡改性，可以从根本上解决贸易背景真实性的问题。

通过 Fabric 相关的技术特性，使得供应链金融系统中的各个参与方从供应商、核心企



业、分销商到物流企业、仓储监管公司、金融机构等,均可运用区块链技术进行信息共享。在 Fabric 中每笔交易的数据均由该交易所有的参与方集体认定、共同维护。这样又解决了多个参与方之间的互信问题。由此可见 Fabric 的技术特性可以有效解决供应链金融系统中存在的一些问题。

## 12.2 用 Fabric 构建供应链金融系统的方法

在上节中我们提出了供应链的常用场景已经遇到的问题,同时也介绍了通过 Fabric 可以解决的这些问题。接下来我们将通过用 Fabric 实现一个具体的场景来说明 Fabric 是如何解决这些问题的。由于供应链金融系统涉及的场景比较多,我们无法一一列举,这么我们选取这些场景中最简单的“应收账款融资”来作为下面演示项目的目标场景。

### 12.2.1 系统的设计

在应收账款融资场景中通常涉及核心企业、供应商、金融机构这三个参与方。我们通过图 12-1 来说明这些参与方是如何通过区块链进行协作的。

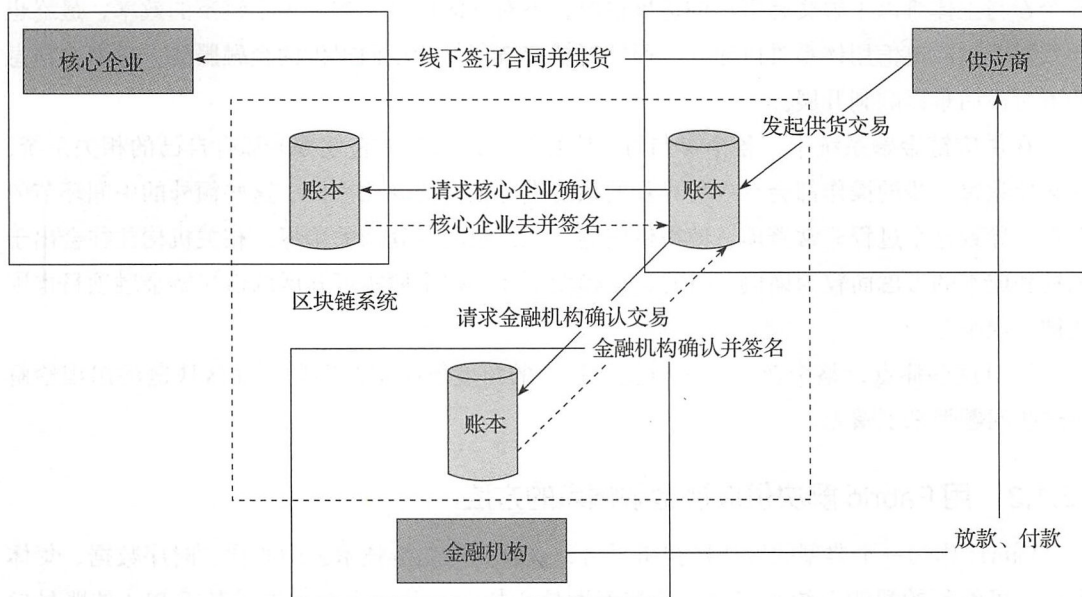


图 12-1 应收账款融资业务场景图

通过图 12-1 我们可以发现,每一笔交易都是所有的参与共同认证方可生效。因此我们将每个参与方抽象成 Fabric 中的一个组织,根据上面的描述我们可以设计出如图 12-2 所示的系统架构图。

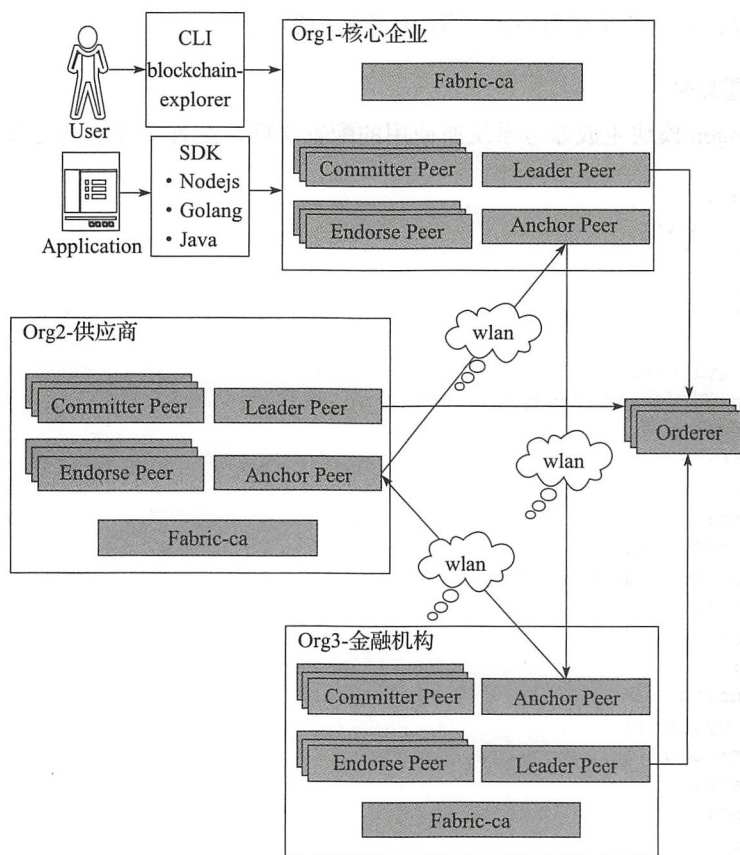


图 12-2 应收账款融资业务场景系统架构图

按照图 12-2 所示的 Fabric 架构图，演示系统中存在三个组织——核心企业、供应商、金融机构，我们按照以下规则给三个组织命名：

表 12-1 供应链金融系统组织信息表

机构名称	组织标识符	组织 ID
核心企业	gyl_org1	GylOrg1MSP
供应商	gyl_org2	GylOrg2MSP
金融机构	gyl_org3	GylOrg3MSP

上面的配置信息非常重要，组织标识符和组织 ID 必须和后面的配置文件一致，读者请留意这里设置的值。

### 12.2.2 系统环境搭建

根据表 12-1 提供的内容，现在可以开始部署 Fabric 的运行环境了。这里我们仅仅给出

对应的配置文件, 具体的步骤可以参考第 10 章的内容。

## 1. 创建配置文件

创建 cryptogen 模块生成账号系统所使用的配置文件, 配置文件的内容如下所示:

```
OrdererOrgs:
  - Name: Orderer
    Domain: qklszzngylgyl.com
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: gyl_org1
    Domain: org1.qklszzngylgyl.com
    Template:
      Count: 3
    Users:
      Count: 4
  - Name: gyl_org2
    Domain: org2.qklszzngylgyl.com
    Template:
      Count: 3
    Users:
      Count: 4
  - Name: gyl_org3
    Domain: org3.qklszzngylgyl.com
    Template:
      Count: 3
    Users:
      Count: 4
```

创建 configtxgen 模块生成系统创始块和 Channel 创始块的配置文件, 配置文件的内容如下所示:

```
Profiles:
  TestOrgsOrdererGenesis:
    Orderer:
      <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
    Consortiums:
      SampleConsortium:
        Organizations:
          - *gyl_org1
          - *gyl_org2
          - *gyl_org3
  TestOrgsChannel:
    Consortium: SampleConsortium
```

```

Application:
  <<: *ApplicationDefaults
  Organizations:
    - *gyl_org1
    - *gyl_org2
    - *gyl_org3

Organizations:

  - &OrdererOrg

    Name: OrdererOrg
    ID: OrdererMSP
    MSPDir: /var/qklszzngyl/crypto-config/ordererOrganizations/qklszzngyl.
com/msp

  - &gyl_org1

    Name: GylOrg1MSP
    ID: GylOrg1MSP
    MSPDir: /var/qklszzngyl/crypto-config/peerOrganizations/org1.qklszzngyl.
com/msp

    AnchorPeers:
      - Host: peer0.org1.qklszzngyl.com
        Port: 7051

  - &gyl_org2

    Name: GylOrg2MSP
    ID: GylOrg2MSP
    MSPDir: /var/qklszzngyl/crypto-config/peerOrganizations/org2.qklszzngyl.
com/msp

    AnchorPeers:
      - Host: peer0.org2.qklszzngyl.com
        Port: 7051

  - &gyl_org3

    Name: GylOrg3MSP
    ID: GylOrg3MSP
    MSPDir: /var/qklszzngyl/crypto-config/peerOrganizations/org3.qklszzngyl.
com/msp

    AnchorPeers:
      - Host: peer0.org3.qklszzngyl.com
        Port: 7051

Orderer: &OrdererDefaults

OrdererType: solo

Addresses:

```



```
- orderer.qklszzngyl.com:7050
```

```
BatchTimeout: 2s
```

```
BatchSize:
```

```
MaxMessageCount: 10
```

```
AbsoluteMaxBytes: 98 MB
```

```
PreferredMaxBytes: 512 KB
```

```
Kafka:
```

```
Brokers:
```

```
- 127.0.0.1:9092
```

```
Organizations:
```

```
Application: &ApplicationDefaults
```

```
Organizations:
```

配置文件创建好之后可以通过 Fabric 的 `configtxgen` 模块生成相关的系统文件, 在第 10 章非常详细地介绍了 Fabric 项目的开发流程, 这里不再复述。将上述两个配置文件严格按照第 10 章流程就可以完成演示用 Fabric 系统的启动。



**注意** 在上述步骤生成账号文件中包含了三个组织: 核心企业、供应商、金融机构的证书, 由于本例中存在三个组织, 因此最少需要启动三个 **Peer** 节点, 启动方法可以参考第 10 章的相关内容。此外, 在按照第 10 章的步骤进行 Fabric 系统部署时一定要注意域名、配置文件的路径、组织 ID 等信息的前后对应关系, 这些因素会导致 Fabric 系统启动失败。

## 2. Chaincode 的编写和发布

根据本例的要求我们编写一个简单的 Chaincode, Chaincode 代码如下所示:

```
package main
```

```
import (
```

```
    "github.com/hyperledger/fabric/core/chaincode/shim"
```

```
    pb "github.com/hyperledger/fabric/protos/peer"
```

```
    "fmt"
```

```
    "encoding/json"
```

```
    "time"
```

```
)
```

```
var asset_name = "asset_name_a"
```

```

type scfinancechaincode struct {}

/**
系统初始化
*/
func (t *scfinancechaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

    fmt.Printf(" Init success      \n " )
    return shim.Success([]byte(" Init success !!!!!!! "))

}

/**
系统 invoke 方法
*/
func (t *scfinancechaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    _,args := stub.GetFunctionAndParameters()

    var opttype = args[0]
    var assetname = args[1]
    var optcontont = args[2]

    fmt.Printf(" parm is  %s  %s  %s   \n " , opttype , assetname , optcontont )

    if opttype == "putvalue" { // 设置

        stub.PutState(assetname, []byte(optcontont))
        return shim.Success( []byte( "success put " + optcontont  ) )

    }else if opttype == "getlastvalue"{ // 取值

        var keyvalue []byte
        var err error
        keyvalue,err = stub.GetState(assetname)

        if( err != nil ){

            return shim.Error(" finad error! ")
        }

        return shim.Success( keyvalue )

    }else if opttype == "gethistroy"{ // 获取交易记录

```

```

        keysIter, err := stub.GetHistoryForKey( assetname );
        if err != nil {
            return shim.Error(fmt.Sprintf("GetHistoryForKey failed. Error accessing
state: %s", err))
        }
        defer keysIter.Close()
        var keys []string
        for keysIter.HasNext() {

            response, iterErr := keysIter.Next()
            if iterErr != nil {
                return shim.Error(fmt.Sprintf("GetHistoryForKey operation failed.
Error accessing state: %s", err))
            }
            // 交易编号
            txid := response.TxId
            // 交易的值
            txvalue := response.Value
            // 当前交易的状态
            txstatus := response.IsDelete
            // 交易发生的时间戳
            txtimesamp :=response.Timestamp
            tm := time.Unix(txtimesamp.Seconds, 0)
            datestr := tm.Format("2006-01-02 03:04:05 PM")

            fmt.Printf(" Tx info - txid : %s value : %s if delete: %t datetime :
%s \n ", txid , string(txvalue) , txstatus , datestr )
            keys = append( keys , txid)

        }

        jsonKeys, err := json.Marshal(keys)

        if err != nil {
            return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
        }

        return shim.Success(jsonKeys)

    }else{

        return shim.Success([]byte("success invok and No operation !!!!!!! "))

    }

}

```

将上述 Chaincode 代码按照第 10 章中的相关步骤部署到已经启动的 Fabric 系统中。

### 12.2.3 客户端开发

本例的客户端代码依然通过 Nodejs 进行开发，首先我们对 Fabric 相关的接口进行封装，封装之后的代码保存在名为 fabricservice.js 的文件中，该源代码文件的内容如下所示：

```
var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
var FabricCAService = require('fabric-ca-client');

var hfc = require('fabric-client');
var log4js = require('log4js');
var logger = log4js.getLogger('Helper');
logger.setLevel('DEBUG');

var tempdir = "/project/ws_nodejs/fabric_sdk_node_studynew/fabric-client-kvs";

let client = new hfc();
var channel = client.newChannel('roberttestchannel12');
var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);
var peer212 = client.newPeer('grpc://172.16.10.212:7051');
channel.addPeer(peer212);
var peer213 = client.newPeer('grpc://192.168.23.213:7051');
channel.addPeer(peer213);
var peer214 = client.newPeer('grpc://192.168.23.214:7051');
channel.addPeer(peer214);

/**
 * 发起一笔交易
 *
 * @returns {Promise.<TResult>}
 */
var sendTransaction = function ( chaincodeid , func , chaincode_args , channel
) {

    var tx_id = null;

    return getOrgUser4Local().then((user)=>{

        tx_id = client.newTransactionID();
        var request = {
```



```

        chaincodeId: "cc_endfinlshed",
        fcn: "invoke",
        args: ["a", "b", "1"],
        chainId: "roberttestchannel12",
        txId: tx_id
    });

    return channel.sendTransactionProposal(request);
} , (err)=>{

    console.log('error', e);

} ).then((chaincodeinvokresult )=>{

    var proposalResponses = chaincodeinvokresult[0];
    var proposal = chaincodeinvokresult[1];
    var header = chaincodeinvokresult[2];
    var all_good = true;

    for (var i in proposalResponses) {

        let one_good = false;
        if (proposalResponses && proposalResponses[0].response &&
            proposalResponses[0].response.status === 200) {
            one_good = true;
            console.info('transaction proposal was good');
        } else {
            console.error('transaction proposal was bad');
        }
        all_good = all_good & one_good;
    }

    if (all_good) {

        console.info(util.format(

            'Successfully sent Proposal and received ProposalResponse: Status -
%s, message - "%s", metadata - "%s", endorsement signature: %s',
            proposalResponses[0].response.status, proposalResponses[0].response.
message,

            proposalResponses[0].response.payload, proposalResponses[0].endorsement
            .signature));

        var request = {
            proposalResponses: proposalResponses,

```

```

        proposal: proposal,
        header: header
    };
    // set the transaction listener and set a timeout of 30sec
    // if the transaction did not get committed within the timeout period,
    // fail the test
    var transactionID = tx_id.getTransactionID();

    return channel.sendTransaction(request);

}

}, (err) => {

    console.log('error', e);
}).then(( sendtransresult ) => {

    return sendtransresult;

}, (err) => {
    console.log('error', e);
});

}

/**
 * 根据 cryptogen 模块生成的账号通过 Fabric 接口进行相关的操作
 *
 * @returns {Promise.<TResult>}
 */
function getOrgUser4Local() {

    // 测试通过 CA 命令行生成的证书依旧可以成功的发起交易
    var keyPath = "/project/fabric_resart/config_demo/org1/186/fabric-user/msp/keystore";
    var keyPEM = Buffer.from(readAllFiles(keyPath)[0]).toString();
    var certPath = "/project/fabric_resart/config_demo/org1/186/fabric-user/msp/signcerts";
    var certPEM = readAllFiles(certPath)[0].toString();

    return hfc.newDefaultKeyValueStore({

        path: tempdir
    })
}

```

## 242 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

    }).then((store) => {
        client.setStateStore(store);

        return client.createUser({
            username: 'user87',
            mspid: 'Org1MSP',
            cryptoContent: {
                privateKeyPEM: keyPEM,
                signedCertPEM: certPEM
            }
        });
    });
};

function readAllFiles(dir) {
    var files = fs.readdirSync(dir);
    var certs = [];
    files.forEach((file_name) => {
        let file_path = path.join(dir, file_name);
        let data = fs.readFileSync(file_path);
        certs.push(data);
    });
    return certs;
}

```

```
exports.sendTransaction = sendTransaction;
```

Web 服务的 Web 框架依然采用 Nodejs 中的 express 框架, 在项目开始之前需要通过 npm 工具安装 express 框架的相关包。安装前请先进入项目的目录, 然后执行以下安装命令:

```
npm install express
```

将 web 服务代码保存在名为 fabricscf.js 文件中, 源代码内容如下所示:

```

var co = require('co');
var fabricservice = require('./fabricservice.js')
var express = require('express');

var app = express();

var channelId = "qklszznscfchannel";
var chaincodeid = "qklszznscfcc";

```

```
// 供应商发起供货交易
```

```

app.get('/sendTransaction1', function (req, res) {

    co( function * () {
        var blockinfo = yield fabricservice.sendTransaction(chaincodeid,"invoke",["putvalue","tans_id1","1"],channelid);
        res.send( "success" );
    })

});

// 核心企业发起确认
app.get('/sendTransaction2', function (req, res) {

    co( function * () {
        var blockinfo = yield fabricservice.sendTransaction(chaincodeid,"invoke",["putvalue","tans_id1","2"],channelid);
        res.send( "success" );
    })

});

// 金融机构审核并放款
app.get('/sendTransaction3', function (req, res) {

    co( function * () {
        var blockinfo = yield fabricservice.sendTransaction(chaincodeid,"invoke",["putvalue","tans_id1","100"],channelid);
        res.send( "success" );
    })

});

// 查询交易记录
app.get('/queryhistory', function (req, res) {

    co( function * () {
        var blockinfo = yield
fabricservice.sendTransaction(chaincodeid,"invoke",["gethistory","tans_id1","-1"],channelid);
        res.send( "success" );
    })

});

// 启动 http 服务
var server = app.listen(3000, function () {

```



```
var host = server.address().address;
var port = server.address().port;

console.log('Example app listening at http://%s:%s', host, port);
});
```

```
// 注册异常处理器
process.on('unhandledRejection', function (err) {
    console.error(err.stack);
});
```

```
process.on('uncaughtException', console.error);
```

现在可以启动这个简单的应用了, 启动应用的命令如下所示:

```
node fabricscf.js
```

通过以下命令完成交易:

```
// 供应商发起供货交易
http://localhost:3000/sendTransaction1
```

```
// 核心企业发起确认
http://localhost:3000/sendTransaction2
```

```
// 金融机构审核并放款
http://localhost:3000/sendTransaction3
```

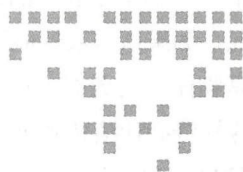
通过以下命令查询交易:

```
// 查询交易内容
http://localhost:3000/queryhistory
```

通过上面的实例, 我们演示了如何将 Fabric 应用到多方参与的业务场景中, 通过使用 Fabric 可以非常方便地解决多方参与过程中信任缺失和信息共享的问题。注意实际的业务场景中业务逻辑远比本例要复杂得多, 建议读者系统开发之前仔细按照第 9 章的内容做好系统的分析工作。

## 12.3 本章小结

本章主要介绍了 Fabric 在供应链金融系统业务场景中的使用, 通过本章内容可以使读者对 Fabric 的使用场景和项目开发流程有更进一步的了解。



## 基于 Fabric 的食品溯源项目实战

区块链技术一个重要的特点是存放在区块链中的数据不可逆不可篡改，同时这些数据又是多方共同拥有的。这些技术特性使得区块链系统能够解决目前数据溯源和确权系统存在的一些问题。Fabric 系统作为典型的区块链技术平台，在继承了区块链基本技术特点的基础上增加了企业应用常见的一些新特性，从而更适用于数据溯源和确权类应用的场景。本章通过一个简单的项目示例，演示如何基于 Fabric 开发一个数据溯源和确权类的系统。

### 13.1 数据溯源的背景知识和痛点

#### 13.1.1 数据溯源的背景知识

在我们的日常生活中，对产品信息的溯源是很常见的应用场景。最普遍的是在很多食品的包装信息上都会有生产日期来验证产品的质量。随着社会发展的进步，这些信息已经不能满足人们的需求，人们需要了解产品更加详细的信息。要做到这有点并不困难，比如我们可以利用现有的技术建立一个从包装打码开始，结合中间配送的过程的商品记录（如现在电商中常用的物流追踪技术）到最后整个消费者系统的下单消费数据的综合系统。但是这些系统提供的数据依然不能保证产品真实的问题，主要缺陷见下文。

#### 13.1.2 数据溯源的痛点

数据溯源领域有如下痛点：

- 数据共享太少：商品溯源的问题可能还要往前去追溯，最好能够将该商品的生产环境

给记录下来。如果是农产品,甚至要记录在生产环节关键细节,比如农药的施用降水量等情况。如果这些数据能够如实记录,对于增加商品的可信度会有很大帮助。另外,不仅仅是物流上的数据,还需要更多的信息录入,比如该商品在整个供应链中流动的信息,这样势必让消费者可以看到完整的参与方数据,以此来增加更多信任背书主体。

- **信息存储中心化:** 这么多信息记录都是在单一的系统里。谁作为中心维护这个账本变成了问题的关键。无论是源头企业保存,还是渠道商保存,由于其自身都是流转链条上的利益相关方,当账本信息不利于其自身时,其很可能选择篡改账本或者谎称账本信息由于技术原因而丢失了。
- **信息存储孤岛化:** 目前主流的系统在整个商品的供应链中,存在信息孤岛问题。通常情况下整个供应链存在多个信息系统,而信息系统之间很难交互,导致信息核对繁琐,数据交互不均衡,最后造成线下需要太多的核对及重复检查才能弥补多个系统交互的问题。另外,由于支付和账期问题而造成的重复审计成本也特别高。

综上所述,区块链具备的信息不可逆、不可篡改的特性以及账本数据共同维护共同确认的特点为溯源、防伪、确权等场景提供了有力的支持。

## 13.2 Fabric 如何优化数据溯源系统

通过前面的内容我们知道传统的中心化技术在产品数据溯源类应用中存在问题,而区块链技术可以有效解决这些问题。Fabric 作为一个典型的区块链技术平台,解决这类问题具有得天独厚的优势。这些优势主要体现在以下几点:

- **可以方便地接入更多参与方** Fabric 可以很容易地让更多的参与方参与进来,共同维护同一个账本,争取尽可能多的商品供应链参与方参与其中。参与方越多,共同维护的数据越多,越容易给消费者带来更多的数据信任背书。
- **克服中心化系统的弊端** Fabric 是一个去中心化的分布式账本,分布式的网络天然克服了中心化系统的各种弊端,同时还能回避人为作恶或者数据意外损失的问题。
- **所有数据共同维护打破信息孤岛** Fabric 可以有效利用众多参与共同维护同一账本的特性,进而打破不同系统间信息孤岛的问题。同时还可以带来支付即结算的清算功能,减少多方重复对账带来的问题和成本,避免溯源过程中成本过高的问题。

## 13.3 Fabric 如何构建数据溯源系统

在上节中我们提出产品溯源存在的问题,同时也介绍了可以通过 Fabric 解决这些问题。接下来我们将基于 Fabric 开发一个简单的溯源系统,说明如何利用 Fabric 开发溯源类的系

统。假设我们打造一款高端的精品鲜奶，那我们就需要对牛奶进行溯源。通过溯源系统，就可以知道每瓶牛奶的全部信息，包括是哪只奶牛产的、奶牛的食物状态、健康状态等信息。我们通过下面的架构图可以详细了解牛奶溯源的业务流程。

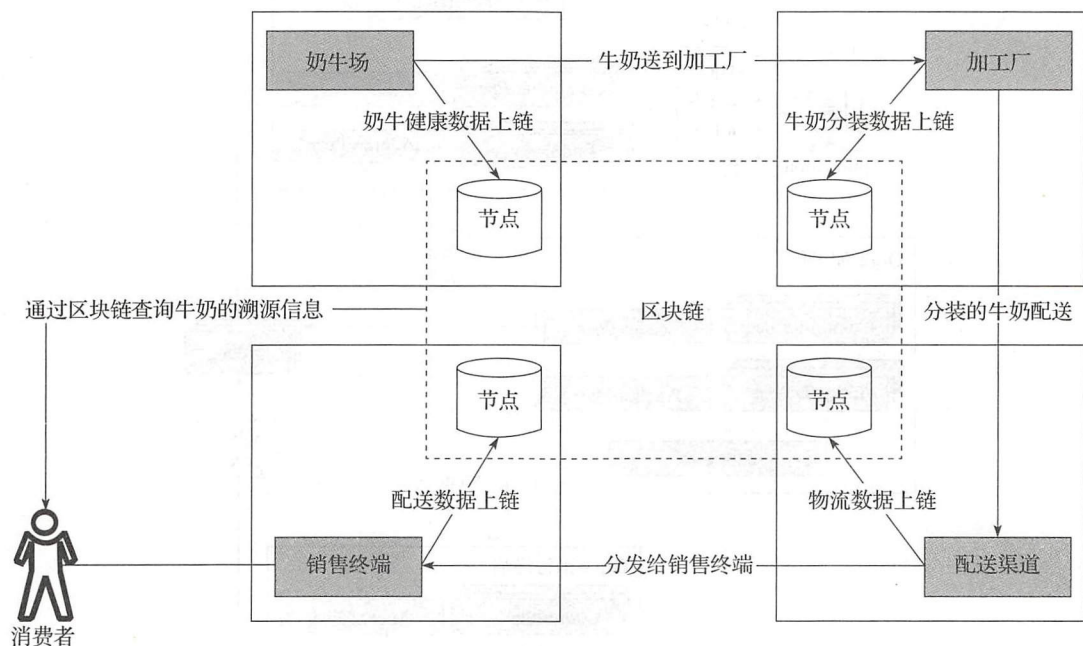


图 13-1 牛奶溯源业务场景图

通过上图我们可以发现牛奶溯源产品的溯源过程中有三个参与方：奶牛场、加工厂、运输渠道，这三方共同参与整个牛奶的溯源过程。我们假设奶牛场对每只奶牛的状态都有精确的记录。每只奶牛每产一定数量的鲜奶都要及时送到加工车间进行包装，加工车间每次用容积为 24 升的罐子从奶牛场取回牛奶，在加工车间每罐 24 升的大包装，会拆分成 1 升的小包装并送到客户。每盒小包装的牛奶的包装盒子上面都会印有相关的编号，通过这些编号可以查询到牛奶的整个生产过程。

根据业务场景图我们可以得出系统的架构，如图 13-2 所示：

按照上面的 Fabric 架构图，牛奶溯源演示系统中存在三个组织，奶牛场、加工厂、销售终端，我们按照以下规则给三个组织命名：

表 13-1 的配置信息非常重要，组织标识符和组织 ID 必须和后面的配置文件一致，在后面的配置过程中请注意这里设置的值。

### 13.3.1 系统环境搭建

根据表 13-1 提供的内容，现在可以开始部署 Fabric 的环境了。这里我们仅仅给出对应



的配置文件, 具体的步骤可以参考第 10 章的内容。

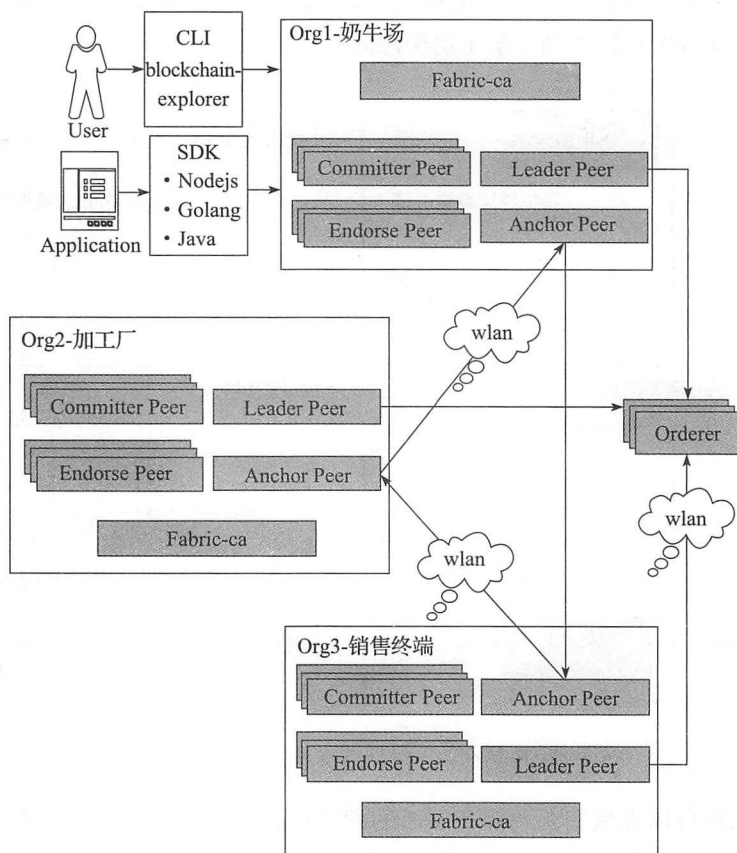


图 13-2 牛奶溯源业务场景架构图

表 13-1 食品溯源系统组织信息表

机构名称	组织标识符	组织 ID
奶牛场	sy_org1	SyOrg1MSP
加工厂	sy_org2	SyOrg2MSP
销售终端	sy_org3	SyOrg3MSP

## 1. 配置演示系统的配置文件

利用 cryptogen 模块生成账号系统的配置文件如下所示：

```
OrdererOrgs:
- Name: Orderer
  Domain: qklszzngsy.com
  Specs:
    - Hostname: orderer
```

```
PeerOrgs:
- Name: gyl_org1
  Domain: org1.qklszzngsy.com
  Template:
    Count: 3
  Users:
    Count: 4
- Name: gyl_org2
  Domain: org2.qklszzngsy.com
  Template:
    Count: 3
  Users:
    Count: 4
- Name: gyl_org3
  Domain: org3.qklszzngsy.com
  Template:
    Count: 3
  Users:
    Count: 4
```

利用 `configtxgen` 模块生成系统创始块和 Channel 创始块的配置文件如下所示:

```
Profiles:

TestOrgsOrdererGenesis:
  Orderer:
    <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
  Consortiums:
    SampleConsortium:
      Organizations:
        - *sy_org1
        - *sy_org2
        - *sy_org3

TestOrgsChannel:
  Consortium: SampleConsortium
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *sy_org1
      - *sy_org2
      - *sy_org3

Organizations:

- &OrdererOrg

  Name: OrdererOrg
  ID: OrdererMSP
```

## 250 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

MSPDir: /var/qklszzngyl/crypto-config/ordererOrganizations/qklszzngsy.
com/msp

- &sy_org1

  Name: SyOrg1MSP
  ID: SyOrg1MSP
  MSPDir: /var/qklszzngyl/crypto-config/peerOrganizations/org1.qklszzngsy.
com/msp

  AnchorPeers:
    - Host: peer0.org1.qklszzngsy.com
      Port: 7051

- &sy_org2

  Name: SyOrg2MSP
  ID: SyOrg2MSP
  MSPDir: /var/qklszzngyl/crypto-config/peerOrganizations/org2.qklszzngsy.
com/msp

  AnchorPeers:
    - Host: peer0.org2.qklszzngsy.com
      Port: 7051

- &sy_org3

  Name: SyOrg3MSP
  ID: SyOrg3MSP
  MSPDir: /var/qklszzngyl/crypto-config/peerOrganizations/org3.qklszzngsy.
com/msp

  AnchorPeers:
    - Host: peer0.org3.qklszzngsy.com
      Port: 7051

Orderer: &OrdererDefaults

  OrdererType: solo

  Addresses:
    - orderer.qklszzngsy.com:7050

  BatchTimeout: 2s

  BatchSize:

    MaxMessageCount: 10
    AbsoluteMaxBytes: 98 MB
    PreferredMaxBytes: 512 KB

  Kafka:

    Brokers:
      - 127.0.0.1:9092
  
```

Organizations:

Application: &ApplicationDefaults

Organizations:

利用上述配置文件按照第 10 章的步骤可以完成 Fabric 系统配置和部署。



**注意** 在上述步骤生成账号文件中包含了三个组织：奶牛场、加工厂、销售终端的证书，由于本例中存在三个组织，因此最少需要启动三个 Peer 节点，启动方法可以参考第 10 章的相关内容。此外，在按照第 10 章的步骤进行 Fabric 系统部署的时候一定要注意域名、配置文件的路径、组织 ID 等信息的前后对应关系，这些因素会导致 Fabric 系统启动失败。

## 2. Chaincode 的编写和发布

根据本例的要求，我们编写三个 Chaincode 来对应提供给不同的组织使用，三个组织的命名如表 13-2 所示：

表 13-2 食品溯源系统 Chaincode 注释名称对应表

机构名称	Chaincode 名称
奶牛场	cc_dairyfarm
加工厂	cc_machining
销售终端	cc_salesterminal

现在我们创建存放 Chaincode 的目录，为每个 Chaincode 创建一个单独的目录，目录名和 Chaincode 的名字是一致的，创建目录的命令如下所示：

```
mkdir -p $GOPATH/src/qklszzl/origin_dairyfarm
mkdir -p $GOPATH/src/qklszzl/origin_machining
mkdir -p $GOPATH/src/qklszzl/origin_salesterminal
```

目录创建完成之后分别创建每个组织对应的 Chaincode。

### 创建组织“奶牛场”需要的 Chaincode

组织“奶牛场”对应的 Chaincode 代码如下所示：

```
package main

import (

    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
    "fmt"
```



## 252 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

        "time"
        "encoding/json"
    )

type dairyfarm struct {}

func (t *dairyfarm) Init(stub shim.ChaincodeStubInterface) pb.Response {

    return shim.Success([]byte("success invok  and Not opter !!!!!!! "))
}

func (t *dairyfarm) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    _,args := stub.GetFunctionAndParameters()

    var opttype = args[0]
    var assetname = args[1]
    var optcontont = args[2]

    fmt.Printf(" parm is  %s  %s  %s  \n " , opttype , assetname , optcontont )

    if opttype == "putvalue" { //设置

        stub.PutState(assetname,[]byte(optcontont))
        return shim.Success( []byte( "success put " + optcontont  ) )

    }else if opttype == "getlastvalue"{ //取值

        var keyvalue []byte
        var err error
        keyvalue,err = stub.GetState(assetname)

        if( err != nil ){

            return shim.Error(" finad error! ")
        }

        return shim.Success( keyvalue )

    }else if opttype == "gethistory"{ //获取交易记录

```

```

keysIter, err := stub.GetHistoryForKey( assetname );

if err != nil {
    return shim.Error(fmt.Sprintf("GetHistoryForKey failed. Error accessing
state: %s", err))
}
defer keysIter.Close()

var keys []string

for keysIter.HasNext() {

    response, iterErr := keysIter.Next()
    if iterErr != nil {
        return shim.Error(fmt.Sprintf("GetHistoryForKey operation failed.
Error accessing state: %s", err))
    }
    // 交易编号
    txid := response.TxId
    // 交易的值
    txvalue := response.Value
    // 当前交易的状态
    txstatus := response.IsDelete
    // 交易发生的时间戳
    txtimesamp := response.Timestamp

    tm := time.Unix(txtimesamp.Seconds, 0)
    datestr := tm.Format("2006-01-02 03:04:05 PM")

    fmt.Printf(" Tx info - txid : %s value : %s if delete: %t datetime :
%s \n ", txid , string(txvalue) , txstatus , datestr )
    keys = append( keys , string(txvalue) + ":" + datestr)

}

jsonKeys, err := json.Marshal(keys)
if err != nil {
    return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
}

return shim.Success(jsonKeys)

}else{

    return shim.Success([]byte("success invok and No operation !!!!!!! "))

}

```

```
}
```

```
func main() {
    err := shim.Start( new( dairyfarm ) )
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}
```

上述代码存储到文件 `origin_dairyfarm.go` 中, 然后将文件保存到目录 `$GOPATH/src/qklszzl/origin_dairyfarm` 中。

### 创建组织“加工厂”需要的 Chaincode

组织“加工厂”对应的 Chaincode 代码如下所示:

```
package main
```

```
import (
```

```
    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
    "fmt"
    "time"
    "encoding/json"
```

```
)
```

```
type machining struct {}
```

```
func (t *machining) Init(stub shim.ChaincodeStubInterface) pb.Response {
    return shim.Success([]byte("success invoked and Not opted !!!!!!! "))
}
```

```
func (t *machining) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    _, args := stub.GetFunctionAndParameters()

    var opttype = args[0]
    var assetname = args[1]
    var optcontent = args[2]

    fmt.Printf(" parm is %s %s %s \n ", opttype , assetname , optcontent )
```

```

if opttype == "putvalue" {

    stub.PutState(assetname, []byte(optcontont))
    return shim.Success( []byte( "success put " + optcontont  ) )

}

}else if opttype == "getlastvalue"{

    var keyvalue []byte
    var err error
    keyvalue,err = stub.GetState(assetname)

    if( err != nil ){

        return shim.Error(" finad error! ")

    }

    return shim.Success( keyvalue )

}

}else if opttype == "gethistory"{ // 获取交易记录

    keysIter, err := stub.GetHistoryForKey( assetname );

    if err != nil {
        return shim.Error(fmt.Sprintf("GetHistoryForKey failed. Error accessing
state: %s", err))
    }
    defer keysIter.Close()

    var keys []string

    for keysIter.HasNext() {

        response, iterErr := keysIter.Next()
        if iterErr != nil {
            return shim.Error(fmt.Sprintf("GetHistoryForKey operation failed.
Error accessing state: %s", err))
        }
        // 交易编号
        txid := response.TxId
        // 交易的值
        txvalue := response.Value
        // 当前交易的状态
        txstatus := response.IsDelete
        // 交易发生的时间戳
    }
}

```



```

        txtimesamp :=response.Timestamp

        tm := time.Unix(txtimesamp.Seconds, 0)
        datestr := tm.Format("2006-01-02 03:04:05 PM")

        fmt.Printf(" Tx info -   txid : %s   value : %s   if delete: %t   datetime :
%s \n ", txid , string(txvalue) , txstatus , datestr )

        keys = append( keys , string(txvalue) + ":" + datestr )

    }

    jsonKeys, err := json.Marshal(keys)
    if err != nil {
        return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
    }

    return shim.Success(jsonKeys)

}

return shim.Success([]byte("success invok and No operation !!!!!!! "))

}

}

func main() {
    err := shim.Start(new( machining ))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}

```

代码存储到文件 `origin_machining.go` 中，然后将文件保存到目录 `$GOPATH/src/qklszzl/origin_machining` 中。

### 创建组织“销售终端”需要的 Chaincode

组织“销售终端”对应的 Chaincode 代码如下所示：

```

/**

Copyright xuehuiit Corp. 2018 All Rights Reserved.

http://www.xuehuiit.com

```



```

QQ 411321681

*/

package main

import (

    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
    "fmt"
    "time"
    "encoding/json"

    "strings"
)

type salesterminal struct {}

/**
    系统初始化
*/
func (t *salesterminal) Init(stub shim.ChaincodeStubInterface) pb.Response {

    _,args := stub.GetFunctionAndParameters()

    var a_parm = args[0]
    var b_parm = args[1]
    var c_parm = args[2]

    fmt.Printf(" parm is  %s  %s  %s  \n " , a_parm , b_parm , c_parm )

    return shim.Success([]byte("success invok  and Not opter !!!!!!! "))

}

/**
    系统 invoke 方法
*/
func (t *salesterminal) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    _,args := stub.GetFunctionAndParameters()

    var opttype = args[0]
    var assetname = args[1]

```



```

var optcontont = args[2]

fmt.Printf(" parm is  %s  %s  %s  \n ", opttype , assetname , optcontont )

if opttype == "putvalue" { // 设置

    stub.PutState(assetname,[]byte(optcontont))
    return shim.Success( []byte( "success put " + optcontont  ) )

}

}else if opttype == "getlastvalue"{ // 取值

    var keyvalue []byte
    var err error
    keyvalue,err = stub.GetState(assetname)

    if( err != nil ){

        return shim.Error(" finad error! ")
    }

    return shim.Success( keyvalue )

}

}else if opttype == "gethistory"{

    keysIter, err := stub.GetHistoryForKey( assetname );

    if err != nil {
        return shim.Error(fmt.Sprintf("GetHistoryForKey failed. Error accessing
state: %s", err))
    }
    defer keysIter.Close()

    var keys []string

    for keysIter.HasNext() {

        response, iterErr := keysIter.Next()
        if iterErr != nil {
            return shim.Error(fmt.Sprintf("GetHistoryForKey operation failed.
Error accessing state: %s", err))
        }
        // 交易编号
        txid := response.TxId
        // 交易的值
        txvalue := response.Value
    }
}

```



```

// 当前交易的状态
txstatus := response.IsDelete
// 交易发生的时间戳
txtimesamp := response.Timestamp

tm := time.Unix(txtimesamp.Seconds, 0)
datestr := tm.Format("2006-01-02 03:04:05 PM")

fmt.Printf(" Tx info -  txid : %s  value :  %s  if delete: %t  datetime :
%s \n ", txid , string(txvalue) , txstatus , datestr )
keys = append( keys , string(txvalue) + ":" + datestr )

}

jsonKeys, err := json.Marshal(keys)
if err != nil {
    return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
}

return shim.Success(jsonKeys)

}else if opttype == "getmilkhistory"{ // 获取交易记录

// 获取销售终端的历史记录

keysIter, err := stub.GetHistoryForKey( assetname );

if err != nil {
    return shim.Error(fmt.Sprintf("GetHistoryForKey failed. Error accessing
state: %s", err))
}
defer keysIter.Close()

var keys []string
var values []string

for keysIter.HasNext() {

    response, iterErr := keysIter.Next()
    if iterErr != nil {
        return shim.Error(fmt.Sprintf("GetHistoryForKey operation failed.
Error accessing state: %s", err))
    }
    // 交易编号
    txid := response.TxId

```





```

// 交易的值
txvalue := response.Value
// 当前交易的状态
txstatus := response.IsDelete
// 交易发生的时间戳
txtimesamp :=response.Timestamp

tm := time.Unix(txtimesamp.Seconds, 0)
datestr := tm.Format("2006-01-02 03:04:05 PM")

fmt.Printf(" Tx info -   txid : %s   value : %s   if delete: %t   datetime :
%s \n ", txid , string(txvalue) , txstatus , datestr )

keys = append( keys , string(txvalue) + ":" + datestr )

values = append( values,string(txvalue))

}

// 获取工厂编号
machiningid := values[0]

// 调用加工厂的 chaincode 获取加工厂的溯源信息
machining_history_parm := []string{"invoke","gethistory",machiningid,"a"}
queryArgs := make([][]byte, len(machining_history_parm))
for i, arg := range machining_history_parm {
    queryArgs[i] = []byte(arg)
}

response := stub.InvokeChaincode("cc_machining",queryArgs,"roberttestchannel12")

if response.Status != shim.OK {
    errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s",
response.Payload)
    fmt.Printf(errStr)
    return shim.Error(errStr)
}

// 获取加工的信息
result := string(response.Payload)

fmt.Printf(" maching info -   result : %s   \n ", result )

var machinginfos []string
if err := json.Unmarshal( []byte(result), &machinginfos); err != nil {

    return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
}

```



```

    }

    for _,v := range machinginfos{

        keys = append( keys,v)

    }

    milid := machinginfos[0]
    fmt.Printf(" mil info - milid : %s \n ", milid )

    milidarr := strings.Split(milid, ":")
    cowid := milidarr[0]

    fmt.Printf(" mil info - cowid : %s \n ", cowid )

    // == 通过奶牛的编号获取奶牛的溯源信息
    cow_parms := []string{"invoke","gethistory",cowid,"a"}
    queryArgs1 := make([][]byte, len(cow_parms))
    for i, arg := range cow_parms {
        queryArgs1[i] = []byte(arg)
    }

    cow_response :=
    stub.InvokeChaincode("cc_dairyfarm",queryArgs1,"roberttestchannel12")

    if cow_response.Status != shim.OK {
        errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s",
        cow_response.Payload)
        fmt.Printf(errStr)
        return shim.Error(errStr)
    }

    cow_result := string(cow_response.Payload)

    fmt.Printf(" cow info - result : %s \n ", cow_result )

    var cowhistorys []string
    if err := json.Unmarshal( []byte(cow_result), &cowhistorys); err != nil {

        return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
    }

```



```

    }

    for _,v1 := range cowhistorys{

        keys = append( keys,v1)

    }


    jsonKeys, err := json.Marshal(keys)
    if err != nil {
        return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
JSON: %s", err))
    }

    return shim.Success(jsonKeys)

}

else{

    return shim.Success([]byte("success invok and No operation !!!!!!!! "))

}

}

func main() {
    err := shim.Start(new(salesterminal))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}
```

上述代码存储到名为 `origin_saleterminal.go` 的文件中，然后将文件保存在目录 `$GOPATH/src/qklszzl/origin_saleterminal` 中。将上面三个 Chaincode 部署到 Fabric 中，部署的过程可以参考第 10 章的内容。

部署的过程中要注意前面过程中创建的 Channel 的名字。

### 13.3.2 客户端开发

现在我们可以编写一个模拟客户端程序发起交易请求，我们还是采用 Node.js 来编写客户端演示程序，整个源代码分为两个源代码文件，一个是对 Fabric API 的封装，还有一个是

web 服务相关的代码。封装 Fabric API 的源代码的文件名为 `fabricservice.js`, web 服务相关的代码为 `mainorigin.js`, 源代码如下所示:

`fabricservice.js` 的源代码:

```
var path = require('path');
var fs = require('fs');
var util = require('util');
var hfc = require('fabric-client');
var Peer = require('fabric-client/lib/Peer.js');
var EventHub = require('fabric-client/lib/EventHub.js');
var User = require('fabric-client/lib/User.js');
var crypto = require('crypto');
var FabricCAService = require('fabric-ca-client');

var hfc = require('fabric-client');
var log4js = require('log4js');
var logger = log4js.getLogger('Helper');
logger.setLevel('DEBUG');

var channelId = "roberttestchannel12";

var tempdir = "/project/ws_nodejs/fabric_sdk_node_studynew/fabric-client-kvs";

let client = new hfc();
var channel = client.newChannel(channelId);
var order = client.newOrderer('grpc://192.168.23.212:7050');
channel.addOrderer(order);

var peer = client.newPeer('grpc://192.168.23.212:7051');
channel.addPeer(peer);

var queryCc = function (chaincodeid , func , chaincode_args ) {

    return getOrgUser4Local().then(( user )=>{

        tx_id = client.newTransactionID();
        var request = {
            chaincodeId: chaincodeid,
            txId: tx_id,
            fcn: func,
            args: chaincode_args
        };
    });
};
```





## 264 ❖ 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

    return channel.queryByChaincode( request , peer );

    }, (err)=>{

        console.log('error', e);

    }).then(( sendtransresult )=>{

        return sendtransresult;

    }, (err)=>{

        console.log('error', e);

    });

}

/**
 * 发起一笔交易
 *
 * @returns {Promise.<TResult>}
 */
var sendTransaction = function ( chaincodeid , func , chaincode_args ) {

    var tx_id = null;

    return getOrgUser4Local().then((user)=>{

        tx_id = client.newTransactionID();
        var request = {

            chaincodeId: chaincodeid,
            fcn: func,
            args: chaincode_args,
            chainId: channelid,
            txId: tx_id
        };

        return channel.sendTransactionProposal(request);

    }, (err)=>{

        console.log('error', e);

    }).then((chaincodeinvokresult )=>{

        var proposalResponses = chaincodeinvokresult[0];

```



```

var proposal = chaincodeinvokresult[1];
var header = chaincodeinvokresult[2];
var all_good = true;

for (var i in proposalResponses) {

    let one_good = false;
    if (proposalResponses[i].response &&
        proposalResponses[i].response.status === 200) {
        one_good = true;
        console.info('transaction proposal was good');
    } else {
        console.error('transaction proposal was bad');
    }
    all_good = all_good & one_good;
}

if (all_good) {

    console.info(util.format(

        'Successfully sent Proposal and received ProposalResponse: Status -
%s, message - "%s", metadata - "%s", endorsement signature: %s',
        proposalResponses[i].response.status, proposalResponses[i].response.
message,
        proposalResponses[i].response.payload, proposalResponses[i].endor-
sement

        .signature));

    var request = {
        proposalResponses: proposalResponses,
        proposal: proposal,
        header: header
    };
    // set the transaction listener and set a timeout of 30sec
    // if the transaction did not get committed within the timeout period,
    // fail the test
    var transactionID = tx_id.getTransactionID();

    return channel.sendTransaction(request);

}

}, (err) =>{

```



```

        console.log('error', e);
    }).then(() => { sendtransresult }) => {

        return sendtransresult;

    }, (err) => {
        console.log('error', e);
    });
}

/**
 *
 * 根据 cryptogen 模块生成的账号通过 Fabric 接口进行相关的操作
 *
 * @returns {Promise.<TResult>}
 */
function getOrgUser4Local() {

    // 测试通过 CA 命令行生成的证书依旧可以成功的发起交易
    var keyPath = "/project/opt_fabric/fabricconfig/crypto-config/peerOrganizations/
org1.robertfabrictest.com/users/Admin@org1.robertfabrictest.com/msp/keystore";
    var keyPEM = Buffer.from(readAllFiles(keyPath)[0]).toString();
    var certPath = "/project/opt_fabric/fabricconfig/crypto-config/peerOrganizations/
org1.robertfabrictest.com/users/Admin@org1.robertfabrictest.com/msp/signcerts";
    var certPEM = readAllFiles(certPath)[0].toString();

    return hfc.newDefaultKeyValueStore({

        path: tempdir

    }).then((store) => {

        client.setStateStore(store);

        return client.createUser({
            username: 'user87',
            mspid: 'Org1MSP',
            cryptoContent: {
                privateKeyPEM: keyPEM,
                signedCertPEM: certPEM
            }
        });
    });
};

function readAllFiles(dir) {

```

```

var files = fs.readdirSync(dir);
var certs = [];
files.forEach((file_name) => {
  let file_path = path.join(dir, file_name);
  let data = fs.readFileSync(file_path);
  certs.push(data);
});
return certs;
}

```

```

exports.sendTransaction = sendTransaction;
exports.queryCc = queryCc;

```

注意修改源代码中的 IP 地址和 channel 的名字。

mainorigin.js 的源代码:

```

var co = require('co');
var fabricservice = require('./fabricservice.js');
var express = require('express');

var app = express();

var cowid = "cow_001";
var machiningid = "machining_001";
var milk_bottle = "milk_bottle_001";

var cow_cc_name = "cc_dairyfarm";
var machining_cc_name = "cc_machining";
var milkbottle_cc_name = "cc_salesterminal";

var channelid = "";

app.get('/init', function (req, res) {

  co( function * () {

    var dairyfarminitresult = yield fabricservice.sendTransaction(cow_cc_
name, "invoke", ["putvalue", cowid, "food"]);
    var machininginitresult = yield fabricservice.sendTransaction(cow_cc_
name, "invoke", ["putvalue", machiningid, cowid]);
    var salesterminalinitresult = yield fabricservice.sendTransaction(cow_
cc_name, "invoke", ["putvalue", milk_bottle, machiningid]);

    for (let i = 0; i < chiancodequeryresutl.length; i++) {

```



## 268 ❖ 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

        res.send( dairyfarminitresult[i].toString( 'utf8' ) );
    }

    }).catch((err) => {
        res.send(err);
    })

});

// 奶牛场的相关操作
app.get('/dairyfarm', function (req, res) {

    co( function * () {

        var parm = req.query.parms;

        var chiancodequeryresutl = yield fabricservice.sendTransaction(cow_cc_
name,"invoke",[ "putvalue" , cowid , parm ]);

        for (let i = 0; i < chiancodequeryresutl.length; i++) {

            res.send( chiancodequeryresutl[i].toString( 'utf8' ) );

        }

    }).catch((err) => {
        res.send(err);
    })

});

// 加工车间的操作
app.get('/machining', function (req, res) {

    co( function * () {

        var parm = req.query.parms;

        var chiancodequeryresutl = yield fabricservice.sendTransaction(machining_
cc_name,"invoke",["putvalue",machiningid,parm]);

        for (let i = 0; i < chiancodequeryresutl.length; i++) {

            res.send( chiancodequeryresutl[i].toString( 'utf8' ) );

        }

    }

```

```
    }).catch((err) => {  
      res.send(err);  
    })  
  });
```

// 销售终端的操作

```
app.get('/salesterminal', function (req, res) {  
  
  co( function * () {  
  
    var parm = req.query.parms;  
  
    var chiancodequeryresutl = yield fabricservice.sendTransaction(milkbottle_  
cc_name,"invoke",["putvalue",milk_bottle,parm]);  
  
    for (let i = 0; i < chiancodequeryresutl.length; i++) {  
  
      res.send( chiancodequeryresutl[i].toString( 'utf8' ) );  
  
    }  
  
    }).catch((err) => {  
      res.send(err);  
    })  
  });  
});
```

// 客户端查询牛奶的历史

```
app.get('/getmilhistory', function (req, res) {  
  
  co( function * () {  
  
    var chiancodequeryresutl = yield fabricservice.queryCc(milkbottle_cc_  
name,"invoke",["getmilhistory",milk_bottle,"a"],)  
  
    for (let i = 0; i < chiancodequeryresutl.length; i++) {  
  
      res.send( chiancodequeryresutl[i].toString( 'utf8' ) );  
  
    }  
  
  }  
});
```

```

    }).catch((err) => {
      res.send(err);
    })
  });

// 启动 http 服务
var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});

// 注册异常处理器
process.on('unhandledRejection', function (err) {
  console.error(err.stack);
});

process.on('uncaughtException', console.error);

```

现在可以启动这个简单的浏览器, 启动浏览器的命令如下所示:

```
node mainorigin.js
```

启动成功之后可以在浏览器中输入以下网址, 可以先执行以下操作了:

### 1. 系统初始化

```
http://localhost:3000/init
```

### 2. 奶牛场执行的操作

```
// 记录奶牛喂食物
```

```
http://localhost:3000/dairyfarm?parms=food
```

```
// 记录奶牛洗澡
```

```
http://localhost:3000/dairyfarm?parms=Takeashower
```

```
// 记录奶牛散步
```

```
http://localhost:3000/dairyfarm?parms=Takeawalk
```

### 3. 加工厂执行的操作

```
// 记录加工厂杀毒
```

```
http://localhost:3000/machining?parms=pasteurisation
```

```
// 记录加灌装
```

```
http://localhost:3000/machining?parms=canned
```

#### 4. 销售终端执行的操作

```
// 记录出厂时间
http://localhost:3000/salesterminal?parms=factory_time
// 记录发货时间
http://localhost:3000/salesterminal?parms=distribution_time
记录有效期
http://localhost:3000/salesterminal?parms=quality_guarantee_period__2017-12-20
```

#### 5. 客户端执行的操作

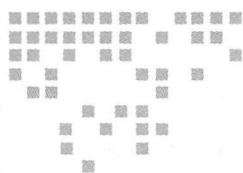
```
http://localhost:3000/getmilhistory
```

在本例中我们演示的 Fabric 如何应用到食品溯源的项目中，特别是在本例中我们演示如何利用多个 Chaincode 组合的形式解决溯源类项目中经常遇到的数据多方认证的问题。通过本实例读者对 Fabric 的特性有了进一步的了解，为后续 Fabric 的技术在实际项目中的应用打下了坚实的基础。

### 13.4 本章小结

本章通过一个基于 Fabric 架构的溯源系统的开发示例，说明了 Fabric 在溯源类系统业务场景的应用。读者通过本章内容的阅读可以继续加深 Fabric 的应用场景和项目开发流程的理解。





## Appendix A

### 附录 A

# 比特币的原理和运行方式

比特币是经典的区块链应用，也是目前最大、最稳当的区块链应用。本章主要介绍比特币的编译和基本的使用方法。通过对本章的阅读，读者会对比特币系统会有一个初步的了解，为后续深入了解比特币的特性做好准备。

## A.1 比特币简介

比特币（BitCoin）的概念最初由中本聪在 2009 年提出，其依托于根据中本聪的思路设计发布的开源软件以及建构其上的 P2P 网络。比特币是一种 P2P 形式的数字货币。P2P 意味着比特币系统是一个去中心化的支付系统。

与大多数货币不同，比特币不依靠特定货币机构发行，它依据特定算法，通过大量的计算产生。比特币经济使用整个 P2P 网络中众多节点构成的分布式数据库来确认并记录所有的交易行为，并使用密码学的设计来确保货币流通过程中各个环节的安全性。P2P 的去中心化特性与算法本身的特点可以确保无法通过大量制造比特币来人为操控币值。基于密码学的设计可以使比特币只能被真实的拥有者转移或支付。这同样确保了货币所有权与流通交易的匿名性。比特币与其他虚拟货币最大的不同是其总数量非常有限，具有极强的稀缺性。比特币系统曾在 4 年内只有不超过 1050 万个，之后的总数量将被永久限制在 2100 万个。

在某些国家和地区可以用比特币来兑现其他物品，比如购买一些虚拟物品，比如网络游戏中的装备。只要有人愿意接受，也可以使用比特币购买现实生活当中的物品。

## A.2 比特币的特征

不同于传统货币，比特币是完全虚拟的。比特币不像我们传统的实体货币可以被人感知，比特币看不见，摸不着，其隐含在收发币的转账记录中。用户只要有证明其控制权的密钥，用密钥解锁，就可以发送比特币。这些密钥通常存储在计算机的数字钱包里。拥有密钥是使用比特币的唯一条件，这让控制权完全掌握在每个人手中。比特币是一个分布式的点对点的网络系统，因此没有“中央”服务器，也没有中央发行机构。比特币是通过“挖矿”产生的，挖矿就是验证比特币交易的同时参与竞赛来解决一个数学问题。任何参与者（比如运行一个完整协议栈的人）都可以做矿工，用他们的电脑算力来验证和记录交易。平均每 10 分钟就有人能验证过去这 10 分钟发生的交易，他将会获得新币作为工作回报。本质上，挖矿就把央行的货币发行和结算功能进行了分布式，用全球化的算力竞争来取代对中央发行机构的需求。

除此之外比特币还具有以下特点：

- 全世界流通：比特币可以在任意一台接入互联网的电脑上管理。不管身处何方，任何人都可以挖掘、购买、出售或收取比特币。
- 专属所有权：操控比特币需要私钥，它可以被隔离保存在任何存储介质。除了用户自己之外无人可以获取。
- 低交易费用：可以免费汇出比特币，但最终对每笔交易将收取约一定比率的交易费以确保交易更快执行。
- 无隐藏成本：作为由 A 到 B 的支付手段，比特币没有烦琐的额度与手续限制。知道对方比特币地址就可以进行支付。
- 跨平台挖掘：用户可以在众多平台上发掘不同硬件的计算能力。

## A.3 比特币技术原理

比特币的分布式特性决定了在比特币系统中不会出现传统货币体系中的中心服务器的概念。由于没有中央权威的存在，信任是比特币系统最大的特性。比特币系统包含账号、交易和矿工这三个核心概念。

### 1. 账号

比特币的账号由数字密钥、比特币地址这两个部分组成。数字密钥实际上并不是存储在网络中，而是由用户生成并存储在一个文件或简单的数据库中，这个文件或数据库称为钱包。每笔比特币交易都需要一个有效的签名才会被存储在区块链。只有有效的数字密钥才能产生有效的数字签名，因此拥有比特币的密钥副本就拥有了该账户的比特币控制权。秘钥通

常是成对出现的, 由一个公钥和一个私钥组成。公钥相当于用户名, 私钥可以理解为密码。比特币的用户通常看不到的数字秘钥, 数字秘钥通常由钱包管理。

## 2. 交易

比特币交易是比特币系统中最重要的部分。根据比特币系统的设计原理, 系统中任何其他部分都是为了确保比特币交易可以被生成、能在比特币网络中传播和通过验证, 并最终添加入全球比特币交易总账簿(比特币区块链)。比特币交易的本质是数据结构, 这些数据结构中含有比特币交易参与者价值转移的相关信息。在比特币系统中用户对比特币的拥有权体现在交易里, 用户拥有的比特币数取决于其拥有的交易数, 所有交易的总和就是用户所拥有的比特币总数。

## 3. 矿工

挖矿是增加比特币货币供应的一个过程, 而矿工可以看作是一个逻辑概念, 可以理解为比特币系统的一个节点。挖矿还可保护比特币系统的安全, 防止欺诈交易, 避免“双重支付”(指多次花费同一笔比特币)。矿工们通过为比特币网络提供算力来换取获得比特币奖励的机会。在 1.2.4 节介绍了常用的共识算法, 其中有一种算法名为 POW(工作量证明), 挖矿的过程就是 POW 算法的一种实现。

# A.4 编译和安装

比特币是基于 C++ 语言开发的, 因此在 Linux 和 UNIX 系统上面进行编译相对比较简单, 但是还是有一些需要注意的地方。我们以 Ubuntu 系统为例给读者演示一下比特币系统的安装过程。

第一步: 安装系统编译所必需的包。

```
sudo apt-get install build-essential libtool autotools-dev autoconf automake
pkg-config libssl-dev libevent-dev bsdmainutils libboost-system-dev libboost-
filesystem-dev libboost-chrono-dev libboost-program-options-dev libboost-test-dev
libboost-thread-dev libboost-all-dev libdb-dev libdb++-dev
```

第二步: 安装比特币编译的必需包。

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:bitcoin/bitcoin
sudo apt-get update
sudo apt-get install libdb4.8-dev libdb4.8++-dev
sudo add-apt-repository ppa:bitcoin/bitcoin
```

第三步: 获取比特币源代码并预编译。

```
mkdir -p /opt/bitcoin
```



```
git clone https://github.com/bitcoin/bitcoin.git
git checkout remotes/origin/0.15
./autogen.sh
./configure
```

上述命令中 `./configure` 可以有参数，具体的参数可以通过 `--help` 参数获取。常用的参数及其注释如下：

- `--disable-wallet`：仅仅作作为比特币的节点启动。
- `--without-gui`：不启动图形界面。

在编译过程中可能可能会遇到这样的错误 “`configure: error: Found Berkeley DB other than 4.8, required for portable wallets`”，此时可以通过参数 `--with-incompatible-bdb` 避免。

第四步：编译和安装。

```
make
make install
```

编译完成之后通过以下命令检查安装是否正确。

```
which bitcoind
```

## A.5 比特币的核心模块及其使用方法

比特币系统源代码编译之后会有三个模块：`bitcoind`、`bitcoin-cli`、`bitcoin-tx`。这个三个模块是比特币的核心组成部分，其作用如表 A-1 所示。

表 A-1 比特币系统模块和注释对应表

模块名称	功能
<code>bitcoind</code>	比特币系统主程序，是一个常驻内存程序，启动之后运行在后台。 <code>bitcoind</code> 负责同步数据、管理交易、管理用户钱包、管理客户端接口等，是整个比特币系统的核心
<code>bitcoin-cli</code>	比特币系统命令行工具。通过 <code>bitcoind-cli</code> 可以管理已经运行的 <code>bitcoind</code> 程序，同时可以进行查询交易、发起交易、创建钱包等操作
<code>bitcoin-tx</code>	比特币交易管理工具。可以创建和管理交易，该工具不是很常用

`bitcoind` 是比特币系统的核心模块，`bitcoind` 通过命令行参数和配置文件来管理比特币的特性。

### A.5.1 快速启动一个比特币系统

下面来启动一个比特币系统。

第一步：创建一个文件夹来存储交易和配置文件。

该文件夹可以在任意位置，但是对大小有一定的要求。截止本书完稿的时候，比特币



系统中所有交易数据文件的大小为 200GB 左右。考虑到比特币系统中交易数量的高速增长, 建议读者在准备存放比特币交易数据文件的文件系统的磁盘时, 确保其剩余空间不小于 300GB。

创建比特币数据存放目录:

```
mkdir -p /opt/bitcoin/bitdata
```

第二步: 创建配置文件。

```
cd /opt/bitcoin/bitdata
```

```
vi bitcoin.conf
```

配置文件内容如下:

```
rest=1
rpcuser=root
rpcpassword=111111
rpcport=33133
rpcallowip=::/0
datadir=/opt/bitcoin/bitdata
```

将上述内容保存到配置文件中, 文件名为 bitcoin.conf。

第三步: 启动比特币系统。

```
bitcoind -daemon -datadir=/opt/bitcoin/bitdata
```

其中, -daemon 参数表示当前启动的比特币系统将在后台运行, 如果是演示启动则可以不添加该参数。

第四步: 利用客户端工具 bitcoin-cli 访问比特币系统。

获取当前比特币系统的区块信息:

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getblockchaininfo
```

第五步: 通过 RESTAPI 访问比特币系统。

通过 RESTAPI 接口的形式访问当前比特币系统的网络信息:

```
curl --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getblockchaininfo", "params": [] }' -H 'content-type: text/plain;' http://root:111111@127.0.0.1:33133/
```

至此已启动了一个完整的比特币系统。



**注意** 上面的方法启动了一个真正的比特币系统。启动完成之后当前节点会自动成为全球比特币系统的一个节点, 并且会从最近的节点中同步数据。经过大概 20 个小时的数据

同步，系统会将比特币的所有交易同步到当前的节点中，此时当前节点会存储比特币从诞生到现在的所有交易信息。如果你不想花费这么长的时间下载数据或者磁盘存储空间有限，那么只需要在 `bitcoind` 的命令行中加上参数 `-testnet`。但是笔者强烈建议读者将比特币的所有交易记录下载的本地。想象一下，如果你可以拥有银行的所有交易记录是怎样的一种体验。这正是区块链和比特币的魅力所在吧，完全公开并且是对等的。

## A.5.2 bitcoind 命令行参数

`bitcoind` 模块通过 `--help` 属性可以查看所有命令行的参数选项。`bitcoind` 命令的参数非常多，可以分一般参数、钱包参数、链接参数、调试测试参数、节点参数、区块链类参数、RPC 服务器参数这七大类。本节将分别介绍这七大类参数及其作用。

### 1. 一般参数

这类参数决定 `bitcoind` 运行方式、版本信息等内容。`bitcoind` 包含如下一般参数：

- `?`：显示帮助信息。
- `version`：显示版本信息。
- `alertnotify`：当收到相关提醒或者看到一个长分叉时执行命令（`%s` 将被替换为消息）。
- `blocknotify`：当最好的货币块改变时执行命令（命令中的 `%s` 会被替换为货币块哈希值）。
- `assumevalid`：如果这个块在链中，则假设它和它的祖先是有效的，并可能跳过它们的脚本验证。
- `conf`：指定配置文件，默认为 `bitcoin.conf`。
- `daemon`：系统启动之后作为守护进程运行在后台。
- `datadir`：指定数据目录。
- `dbcache`：设置数据库缓存大小，单位为兆字节（MB），默认为 25。
- `loadblock`：启动时从 `blk000???.dat` 文件导入数据块。
- `maxorphantx`：内存中保留最大交易数，默认为 100。
- `maxmempool`：保持交易内存池的最大值，默认为 300。
- `mempoolexpiry`：内存池中保留交易最大时间，默认为 72。
- `persistmempool`：系统关闭时是否保存内存池，默认为 1。
- `blockreconstructionextratxn`：交易在内存中的快照数，默认为 100。
- `par`：设置脚本验证的程序，取值范围为  $-2 \sim 16$ ，其中 0 表示自动， $<0$  表示保留自由的核心，默认值为 0。

- pid: 指定 pid (进程 ID) 文件, 默认为 bitcoind.pid。
- prune: 通过修剪 (删除) 旧数据块减少存储需求。此模式将禁用钱包支持, 并与 -txindex 和 -rescan 不兼容。
- reindex-chainstate: 从当前位置重建索引。
- reindex: 启动时重新为当前的 blk000???.dat 文件建立索引。
- sysperms: 创建系统默认权限的文件, 而不是 rnsask 077, 只在关闭钱包功能时有效。
- txindex: 维护一份完整的交易索引, 用于 getrawtransaction RPC 调用, 默认值为 0。

## 2. 链接参数

这类参数主要在网络中使用, bitcoind 包含如下链接参数:

- addnode: 添加一个节点, 并尝试保持与该节点的连接。
- banscore: 与行为异常节点断开连接的临界值, 默认为 100。
- bantime: 重新允许行为异常节点连接所间隔的秒数, 默认为 86400。
- bind: 绑定指定的 IP 地址开始监听。IPv6 地址应使用 [host]:port 格式。
- connect: 仅连接到这里指定的节点。
- discover: 发现自己的 IP 地址 (默认: 监听并且无 -externalip 或 -proxy 时为 1)。
- dns: addnode 允许查询 DNS 并连接。
- dnsseed: 使用 DNS 查找节点, 默认为 1。
- externalip: 制定自己的公共地址。
- forcednsseed: 始终通过 DNS 查询节点地址, 默认为 0。
- listen: 接受来自外部的连接, 默认为 1。
- listenonion: 自动创建隐藏服务数, 默认为 1。
- maxconnections: 最多维护连接节点数, 默认为 125。
- maxreceivebuffer: 最大每连接接收缓存字节, 默认为 10000。
- maxsendbuffer: 最大每连接发送缓存字节, 默认为 10000。
- maxtimeadjustment: null。
- onion: 通过 Tor 在隐藏服务连接节点时使用不同的 SOCKS5 代理, 默认为 proxy。
- onlynet: 连接指定网络中的节点 (IPv4、IPv6 或 onion)。
- permitbaremultisig: 是否转发非 P2SH 格式的多签名交易, 默认为 1。
- peerbloomfilters: 支持利用布隆过滤器过滤区块和交易, 默认为 1。
- port: 监听 <端口> 上的连接, 默认为 8333, 测试网络 testnet 为 18333。
- proxy: 通过 Socks4 代理连接指定网络中的节点 (可选类型为: ipv4、ipv6 或 onion)。
- proxyrandomize: 为每个代理连接随机化凭据。这将启用 Tor 流隔离, 默认为 1。

- seednode: 本地服务器和区块中制定服务器的时间差。本地的时间观可能受到向前或向后的同龄人的影响。默认为 4200 秒。
- timeout: 设置连接超时, 单位为毫秒。
- torcontrol: 控制端口, 默认为 127.0.0.1:9051。
- torpassword: 控制密码, 默认值是空。
- whitebind: 当前节点绑定的 IP 地址。
- whitelist: 服务器的白名单列表。
- maxuploadtarget: 设置数据带宽上限, 单位为 MiB/24h, 默认值为 0。

### 3. 钱包参数

这类参数主要负责钱包相关的配置信息, bitcoind 包含如下钱包参数:

- disablewallet: 不要加载钱包和禁用钱包的 RPC 调用。
- keypool: 设置密匙池的大小, 默认为 100。
- fallbackfee: 当交易估算没有足够数据时, 该交易费 (BTC/kB) 将被使用, 默认为 0.0002。
- discardfee: 每笔交易的费用。
- mintxfee: 交易创建时, 小于该交易费 (BTC/kB) 的被认为是零交易费, 默认为 0.00001。
- paytxfee: 发送的交易每 KB 的手续费。
- rescan: 重新扫描货币区块链以查找钱包丢失的交易。
- salvagewallet: 启动时尝试从破坏的钱包文件 wallet.dat 中恢复私钥。
- spendzeroconfchange: 付款时允许使用未确认的零钱, 默认为 1。
- txconfirmtarget: 如果未设置交易费用, 则自动添加足够的交易费以确保交易在平均  $n$  个数据块内被确认, 默认为 2。
- usehd: 是否使用分层的密钥生成方式, 只有在钱包创建 / 第一次启动时生效, 默认为 1。
- upgradewallet: 将钱包升级到最新的格式。
- wallet: 指定钱包文件 (数据目录内), 默认为 wallet.dat。
- walletbroadcast: 钱包广播事务处理, 默认为 1。
- walletnotify: 当最佳区块变化时执行命令 (命令行中的 %s 会被替换成区块 Hash 值)。
- zapwallettxes: 删除钱包的所有交易记录, 且只有用 rescan 参数启动客户端才能重新取回交易记录 (1 表示保留交易元数据, 如账户所有者和支付请求信息; 2 表示不保留交易元数据)。



#### 4. 调试测试参数

这类参数主要用于系统的调试和测试, bitcoind 包含如下调试、测试类参数:

- uacomment: 附加注释到 User Agent 字符串。
- debug: 输出额外的调试信息。
- debugexclude: 过滤相类别的调试信息, 使得系统便于调试。
- help-debug: 显示调试相关的选项。
- logips: 在调试输出中包含 IP 地址, 默认为 0。
- logtimestamps: 调试信息前添加时间戳。
- maxtxfee: 最大单次转账费用 (BTC), 设置太低可能会导致大宗交易失败, 默认为 0.10。
- printtoconsole: 发送跟踪 / 调试信息到控制台而不是 debug.log 文件。
- shrinkdebugfile: 客户端启动时压缩 debug-log 文件 (默认 no-debug 模式时为 1)。
- testnet: 使用测试网络。

#### 5. 节点参数

这类参数主要用于节点的设置, bitcoind 包含如下节点类参数:

- bytespersigop: 在中继和挖矿时, 交易中每个 sigop 的最小字节数, 默认为 20。
- datacarrier: 是否接受中继和挖矿的带外交易, 默认为 1。
- datacarriersize: 交易数据包的大小, 默认为 83。
- mempoolreplacement: 启用内存池交易替换, 默认为 1。
- minrelaytxfee: 当转发、挖矿和交易创建时, 小于该设置的交易费 (BTC/kB) 被认为是 0, 默认为 0.00001。
- whitelistrelay: 非转发交易模式下也接受转发从白名单节点收到的交易, 默认为 1。

#### 6. 区块链类参数

这类参数主要用于区块相关的设置, bitcoind 包含如下区块相关参数:

- blockmaxweight: 设置区块链的最大宽度。
- blockmaxsize: 设置最大区块大小, 默认为 750000, 单位为字节。
- blockmintxfee: 区块链中存储交易的费率。

#### 7. RPC 服务器相关参数

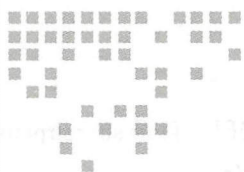
这类参数主要用于 RPC 接口相关的设置, bitcoind 包含如下 RPC 服务器相关参数:

- server: 允许接受 JSON-RPC 的命令。
- rest: 接受公共 REST 请求, 默认为 0。
- rpcbind: 绑定到指定地址并监听 JSON-RPC 连接。IPv6 使用 “[主机]: 端口” 格式。该选项可多次指定, 默认为绑定到所有接口。

- `rpccookiefile`: 验证 cookie 的位置, 默认为数据目录。
- `rpcuser`: JSON-RPC 连接使用的用户名。
- `rpcpassword`: JSON-RPC 连接使用的密码。
- `rpcauth`: JSON-RPC 连接时用的用户名和 Hash 密码。目录 `share/rpcuser` 下有一个权威的 Python 脚本可以使用。这个选项可以配置多次。
- `rpcport`: JSON-RPC 连接所监听的 < 端口 >, 默认为 8332。
- `rpccallowip`: 允许来自指定地址的 JSON-RPC 连接, 可以绑定地址或者地址段。
- `rpcthreads`: 设置 RPC 服务线程数, 默认为 4。

## A.6 本章小结

本章主要介绍了比特币系统的基本原理和使用方法, 通过本章内容读者能对比特币的原理和使用方法有大概的认识, 为后续章节的阅读打下基础。



## Appendix B

### 附录 B

## 比特币的 bitcoin-cli 模块详解

bitcoin-cli 是比特币系统的核心模块，本章将详细介绍 bitcoin-cli 中的命令选项及其使用方法。本章最后会通过一个交易转账的例子来说明如何同通过 bitcoin-cli 模块进行实际操作。

### B.1 bitcoin-cli 模块常用命令

bitcoin-cli 是比特币系统的核心模块。通过 bitcoin-cli 可以有效管理比特币系统，比特币系统提供的所有功能（包括 RESTAPI 接口相关的功能）及绝大多数操作都可以通过 bitcoin-cli 模块来完成管理。需要指出的是，bitcoin-cli 目前只支持管理本机的比特币系统，即 bitcoin-cli 模块只能和比特币系统运行在同一台机器中。

#### B.1.1 bitcoin-cli 初探

##### 1. 运行 bitcoin-cli 的必要条件

首先我们通过几个常用的命令展示一下 bitcoin-cli 模块的功能特性。bitcoin-cli 模块运行的时候需要一些命令行选项，这些命令行选项中有一个特别重要——`-datadir`。`-datadir` 选项的值为当前比特币系统的运行目录。本例中 `-datadir` 的值采用 A.2.1 节中采用的配置文件中的 `datadir` 属性的值，因此本例中 bitcoin-cli 模块演示用的命令，必须基于 A.5.1 节中采用的配置文件启动 bitcoind 模块。在执行的下面的示例命令之前确保 bitcoind 模块已经启动。

## 2. 典型的 bitcoin-cli 命令

bitcoin-cli 模块的命令有很多，这里我们介绍几个常见的 bitcoin-cli 命令，通过这些命的调用方式，让读者对 bitcoin-cli 模块有一个初步了解。

### (1) 获取区块信息

通过该命令可以获取当前比特币系统的区块信息，该命令的格式如下：

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getblockchaininfo
```

### (2) 获取网络信息

通过该命令可以获取当前比特币系统的网络信息，该命令的格式如下：

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getblockchaininfo
```

### (3) 获取当前节点的钱包信息

通过该命令可以获取当前与比特币系统钱包相关的信息，该命令的格式如下：

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getwalletinfo
```

### (4) 根据区块链高度获取相应区块的 Hash 值

通过该命令可以根据指定区块的区块号来获取该区块的 Hash 值，该命令的格式如下：

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getblockhash 0
```

### (5) 获取区块详细信息

通过该命令可以根据某个区块的 Hash 值获取该区块的详细信息，该命令的格式如下：

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getblock 0000000000019d6689c085ae1658  
31e934ff763ae46a2a6c172b3f1b60a8ce26f
```

getblock 方法的参数是区块链的 Hash 值，可以通过 getblockhash 方法获取区块链的 Hash 值。

## B.1.2 bitcoin-cli 的命令及其选项

bitcoin-cli 模块提供了一组命令及其相应的命令选项来执行不行的功能，下面将分别介绍这些命令和选项及其相关的子命令。

### 1. bitcoin-cli 的命令选项

bitcoin-cli 的命令选项分为两大类，一般选项和 RPC 相关的选项。

#### (1) 一般选项

- ?：显示命令帮助信息。



- conf: 配置文件信息。
- datadir: bitcoind 数据文件目录。

## (2) 区块链和 RPC 相关选项

- testnet: 使用测试链。
- regtest: 使用测试链, 在进行回归测试的时候使用。
- name: 命名参数, 默认为 false。
- rpcconnect: RPC 命令发出节点的 IP, 默认为 127.0.0.1。
- rpcport: 连接 JSON-RPC 服务的端口。
- rpcwait: 等待 RPC 服务器启动。
- rpcuser: JSON-RPC 的用户名。
- rpcpassword: JSON-RPC 的密码。
- rpcclienttimeout: 请求 JSON-RPC 的超时时间。
- stdin: 额外的参数。
- rpcwallet: 允许访问的 IP 地址。默认是只能本地访问; 如果想要在其他机器访问, 那么可以配置其他机器的地址; 如果想要本机和和其他机器都可以访问, 可以设置为 -rpccallowip=::/0。

## 2. bitcoin-cli 的相关命令

通过 bitcoin-cli 的命令可以对当前比特币系统进行相关操作, 比如关闭系统、发起交易、管理钱包等。通过下面的命令可以获取所有的 bitcoin-cli 模块的命令列表。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata help
```

bitcoin-cli 的命令分为区块链、系统控制、创建、挖矿、网络、交易、辅助、钱包等八个部分。

### (1) bitcoin-cli 模块中区块链相关的命令

- getbestblockhash: 获取主链中高度最大的区块的 Hash 值。
- getblock: 根据区块编号获取区块信息。
- getblockchaininfo: 获取区块链信息。
- getblockcount: 获取当前系统中最长链的区块数。
- getblockhash: 根据区块的 Hash 值获取区块的详细信息。
- getblockheader: 根据指定的索引返回对应区块的头部信息。
- getchaintips: 获取包括分叉链在内的所有区块链的最大区块信息。
- getdifficulty: 获取当前的谜题的难度 (最低难度的倍数)。
- getmempoolancestors: 获取内存池对应 Hash 的信息, 正序排列。

- getmempooldescendants: 获取内存池对应 Hash 的信息, 逆序排列。
- getmempoolentry: 返回指定交易的内存数据。
- getmempoolinfo: 返回内存池信息。
- getrawmempool: 获取内存池中还没有打包的交易。
- gettxout: 取得未动用的交易输出的详细信息。
- gettxoutproof: 返回某个 txid 在某个块的证据。
- gettxoutsetinfo: 获取未动用的交易统计信息。
- verifychain: 验证区块链数据库。
- verifytxoutproof: 验证 gettxoutproof 命令返回的证据。

#### (2) bitcoin-cli 模块中系统控制相关命令

- getinfo: 返回一个包含当前客户端各种状态信息的对象。
- getmemoryinfo: 返回容器中的内存使用信息。
- help: 帮助服务。获得命令列表, 或者指定命令的帮助。
- stop: 停止比特币客户端服务。
- uptime: 返回服务器的总运行时间。

#### (3) bitcoin-cli 模块中系统创建相关命令

- generate: 立即生成  $x$  个块 (仅用于回归测试模式)。
- generatetoaddress: 立即生成  $x$  个块并发的向地址  $y$  (仅用于回归测试模式)。

#### (4) bitcoin-cli 模块中挖矿相关命令

- getblocktemplate: 根据区块链和相关的构造一个有效的块。
- getmininginfo: 返回当前的矿工信息。
- getnetworkhashps: 获取估算的挖矿 Hash 算力。
- prioritisetransaction: 提高挖矿时的交易被打包的优先级。
- submitblock: 向网络提交新生成的块。

#### (5) bitcoin-cli 模块中网络相关命令

- addnode: 新增节点。
- clearbanned: 清理被禁的 IP。
- disconnectnode: 立刻从指定节点断开。
- getaddednodeinfo: 获取新增加节点信息。
- getconnectioncount: 取得当前节点与其他节点的连接数。
- getnettotals: 获取网络流量统计信息。
- getnetworkinfo: 获取网络信息。
- getpeerinfo: 获取和当前节点连接的其他节点的信息。

- listbanned: 列出所有被禁用的 IP。
- ping: 发送 ping 命令。
- setban: 尝试在禁用列表中加入或删除节点。
- setnetworkactive: 激活或者关闭 P2P 网络。

#### (6) bitcoin-cli 模块中交易相关命令

- combinerawtransaction: 将多个部分签名交易合并为一个交易。
- createrawtransaction: 创建原始交易。
- decoderawtransaction: 将交易的二级制信息解析成 JSON 格式。
- decodescript: 解码脚本。
- fundrawtransaction: 向 createrawtransaction 创建的交易里添加 input 直到满足。
- getrawtransaction: 取得原始交易信息, 返回指定的交易 ID。
- sendrawtransaction: 发布原始交易(序列化的十六进制编码)到本地节点和网络。
- signrawtransaction: 对原始交易签名并返回签名后的交易信息。

#### (7) itcoin-cli 模块中辅助相关命令

- createmultisig: 创建一个多重签名的地址, 并返回一个 JSON 对象。
- estimatefee: 评估达到  $n$  个块确认的交易费。
- signmessagewithprivkey: 用私钥签名消息。
- validateaddress: 验证一个地址是否有效。
- verifymessage: 验证签名后信息是否与指定地址的私钥签名的信息一致。

#### (8) bitcoin-cli 模块中钱包相关命令

- abandontransaction: 启用交易, 从而使交易的输入再次变得可用。
- addmultisigaddress: 在钱包里添加一个多重签名地址。
- addwitnessaddress: 添加隔离认证地址。
- backupwallet: 备份钱包中的数据到文件, 默认文件为 wallet.dat。
- dumpprivkey: 导出钱包地址对应的私钥。需要未锁定钱包。
- dumpwallet: dump 钱包中的数据到指文件中。
- getaccount: 返回指定地址相关联的账户。
- getaccountaddress: 返回指定账户的款地址, 每次执行都会为指定账户创建一个新的地址。
- getaddressesbyaccount: 返回指定账户关联的所有地址列表。
- getbalance: 取得账户余额。
- getnewaddress: 创建并返回一个新的地址, 用于接收付款。
- getrawchangeaddress: 生成一个找零地址。

- `getreceivedbyaccount`: 取得账户收款金额。
- `getreceivedbyaddress`: 返回指定地址上收到的交易总金额。
- `gettransaction`: 根据交易的 Hash 值返回交易详情。
- `getunconfirmedbalance`: 获取未确认的余额。
- `getwalletinfo`: 获取钱包信息。
- `importaddress`: 导入地址。
- `importprivkey`: 导入外部私钥到当前钱包中。
- `importpubkey`: 导入公钥。
- `importwallet`: 从 `backupwallet` 命令备份的文件中恢复钱包数据。
- `keypoolrefill`: 重新填满密钥池。需要未锁定钱包。
- `listaccounts`: 获取当前钱包的地址列表。
- `listaddressgroupings`: 获取钱包上的所有地址信息。
- `listlockunspent`: 列出锁定的暂时未动用的交易输出列表。
- `listreceivedbyaccount`: 列出账户的收款账户信息，返回一个数组对象。
- `listreceivedbyaddress`: 列出地址的收款信息地址，返回一个数组对象。
- `listsinceblock`: 列出指定块之后的交易信息。
- `listtransactions`: 返回指定账户不包含前次的最近交易。如果未指定账户则返回所有账户的最近交易。
- `listunspent`: 返回钱包中未动用交易输入的数组。
- `lockunspent`: 锁定未动用输出交易，更新暂时未动用的交易输出列表。
- `move`: 转账账户。
- `removeprunedfunds`: 从钱包中删除指定交易。
- `sendfrom`: 从账户付款交易。
- `sendmany`: 向多个地址付款交易。
- `sendtoaddress`: 付款交易。
- `setaccount`: 将地址关联到指定账户。如果该地址已经被关联到指定账户，则将创建一个新的地址与该账户关联。
- `settxfee`: 设定交易费。交易交易费是一个四舍五入至小数点后 8 位的实数。
- `signmessage`: 用地址的私钥对信息进行数字签名。需要未锁定钱包。
- `walletlock`: 从内存中删除钱包的加密 KEY，锁定钱包。调用此方法后，您将需要再次调用 `walletpassphrase` 方法，才能够调用其他需要未锁定钱包的方法。
- `walletpassphrase`: 钱包解锁。把钱包的密码存储在内存中持续的时间。
- `walletpassphrasechange`: 修改钱包密码。



## B.2 bitcoin-cli 发起交易

本节中我们将利用 bitcoin-cli 模块来完成一个完整的比特币交易。为了体验真实性,本例中的交易运行在真实的比特币系统里面。每笔交易都是真实的比特转账,因此会产生相关的费用,这些费用都需要用真实的比特币来支付。如果读者没有相关的资源可以通过测试网络来完成这些操作。在测试网络中进行操作只需要在下面的所有命令中增加参数 -regtest 即可。

在执行下面的操作之前,请确保整个系统已经按照 A.5.1 节中的步骤顺利启动 bitcoind 模块并且已经同步了完整的交易数据(如果是测试模式,则可以忽略该步骤)。

通过 bitcoin-cli 模块完成一笔交易需要如下步骤。

第一步:对钱包进行加密并设置密码。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata encryptwallet a123456
```

上述命令中的“a123456”为钱包的密码,读者在演示的时候可以设置任意密码,但是一定要记住,后面会需要这个密码。

第二步:创建账号地址。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata getnewaddress
```

上述命令需要执行三次,需要牢记其中涉及的这三个地址。本例中的三个地址如下:

```
129Kz4HtCCsUX1nJynSxVaXH6S3NQCxVyD  
18kH4DYyEjoyBLXWaaXBHTeDsn7CBAskfD  
15xc46B1PfQG6CWUTNfjTmqoUu2722qVmJ
```

这三个地址是真实的比特币钱包地址,里面依然有比特币,读者可以通过网站 <https://btc.com> 来查询作者验证的记录。特别要注意的是,读者一定要用第二步生成的地址来完成后面的操作。

第三步:查询本地钱包的余额。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata listunspent
```

正常情况下应该是没有任何记录的,因为本地的钱包刚刚生成而且没有任何交易记录。

第四步:给本地钱包转账。

如果读者是在测试链中执行,那么请忽略这一步直接执行后面的步骤,但是应在下面的命令中增加参数 -regtest。如果读者准备在真实的比特币系统中执行下面的步骤,可以通过其他渠道给第二步生成的地址中的其中一个地址转入比特币。主意:转账成功之后等 10 ~ 20 分钟后才能在本地的系统发现这些交易数据。执行第三步的命令后在作者的本地系统中显示的信息如下:

```

bitcoin-cli -datadir=/opt/bitcoin/bitdata listunspent

[
  {
    "txid": "c2892f95603384d12cb92cd0a5cf745b48cb5f6f5881c2e664a390f3a8ce381d",
    "vout": 0,
    "address": "15xc46B1PfQG6CWUTNfjTmqoUu2722qVmJ",
    "account": "",
    "scriptPubKey": "76a9143664a45b9c256ac1c0b160472ce26c69baa6e0a088ac",
    "amount": 0.00526000,
    "confirmations": 7656,
    "spendable": true,
    "solvable": true,
    "safe": true
  },
  {
    "txid": "c2892f95603384d12cb92cd0a5cf745b48cb5f6f5881c2e664a390f3a8ce381d",
    "vout": 1,
    "address": "1L5Fmma1mHXcZH1cYJbWrrF93GQRJRhm78",
    "account": "",
    "scriptPubKey": "76a914d138627a09cffdc006ef51c5c33e0464a46e52ad88ac",
    "amount": 0.00030000,
    "confirmations": 7656,
    "spendable": true,
    "solvable": true,
    "safe": true
  },
  {
    "txid": "0a739b9f33380ec7cb2eb3b9179928c2146b4fc3a54a7da1816ba004111b79db",
    "vout": 1,
    "address": "18kH4DYyEjoyBLXWaaXbHTeDsn7CBAskfD",
    "account": "",
    "scriptPubKey": "76a91454f82b54c82904451023ad3659f448f2ba5d963d88ac",
    "amount": 0.00001000,
    "confirmations": 8551,
    "spendable": true,
    "solvable": true,
    "safe": true
  }
]

```

如果没有显示记录，可能是交易记录还没有同步过来，不要着急，稍等一会即可。

#### 第五步：创建交易。

比特的交易基于 UTXO 原则，又称为未花费过的交易输出。每一笔交易都有来源和输出，创建一笔交易之前首先要确定来源。通过命令 `bitcoin-cli -datadir=/opt/bitcoin/bitdata listunspent` 我们可以获取当前钱包余额，这些余额实际上是当前钱包收到的转账交易。我们从这些交易选择一个满足我们要求的交易来作为来源交易。创建交易的命令如下所示：

## 290 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata createrawtransaction '[{"txid" :
"c2892f95603384d12cb92cd0a5cf745b48cb5f6f5881c2e664a390f3a8ce381d", "vout" : 0}]'
'{"129Kz4HtCCsUX1nJynSxVaXH6S3NQCxVyD": 0.00521000, "18kH4DYyEjoyBLXWaaXbHTeDsn7CBA
skfD": 0.00004}'
```

上面命令涉及的参数如下:

- txid: 来源交易的编号。
- vout: 来源交易中的 vout 值。
- 最后一个中括号中的内容是交易的输出。

上面命令可以理解为: 比特地址 15xc46B1PfG6CWUTNfjTmqoUu2722qVmJ 中存有 0.00526000 个比特币, 我们发起一笔交易, 把这个 0.00526 个比特币分布转给两个地址, 其中 18kH4DYyEjoyBLXWaaXbHTeDsn7CBAAskfD 获取了 0.00521 个比特币, 18kH4DYyEjoyBLXWaaXbHTeDsn7CBAAskfD 获取了 0.00004 个比特币, 剩下的 0.00001 个比特币作为手续费奖励给矿工。

上诉命令的结果如下:

```
02000000011d38cea8f390a364e6c281586f5fcb485b74cfa5d02cb92cd1843360952f89c20000
000000ffffffff0228f307000000000001976a9140c8b4ce0b33d041a3e114b659978fd2646774a9688
aca00f00000000000001976a91454f82b54c82904451023ad3659f448f2ba5d963d88ac00000000
```

第六步: 打开钱包准备转账。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata walletpassphrase a123456 3600
```

第七步: 对交易进行签名。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata signrawtransaction 02000000011d38ce
a8f390a364e6c281586f5fcb485b74cfa5d02cb92cd1843360952f89c20000000000ffffffff0228f3
07000000000001976a9140c8b4ce0b33d041a3e114b659978fd2646774a9688aca00f0000000000019
76a91454f82b54c82904451023ad3659f448f2ba5d963d88ac00000000
```

上面命令涉及的参数

signrawtransaction: 该参数的值为第五步的执行结果。

上述命令执行的结果如下:

```
{
  "hex": "02000000011d38cea8f390a364e6c281586f5fcb485b74cfa5d02cb92cd1843360
952f89c20000000006a47304402205ea2cec0c17a931644e5e7b74c4ec2105bbc064fbae138a6e0b559
c317231d3b02204e5ebc31f190e419733729b0b9162d00baa76fa59940017acffea7f32be8bd5c0121
021c296178856ff484cb6805556190332983b07673066592bca4398e3d43f4a49bfffffffff0228f307
000000000001976a9140c8b4ce0b33d041a3e114b659978fd2646774a9688aca00f000000000001976
a91454f82b54c82904451023ad3659f448f2ba5d963d88ac00000000",
  "complete": true
}
```



第八步：发送交易。

```
bitcoin-cli -datadir=/opt/bitcoin/bitdata sendrawtransaction 02000000011d38
cea8f390a364e6c281586f5fcb485b74cfa5d02cb92cd1843360952f89c2000000006a47304402205e
a2cec0c17a931644e5e7b74c4ec2105bbc064fbae138a6e0b559c317231d3b02204e5ebc31f190e419
733729b0b9162d00baa76fa59940017acffea7f32be8bd5c0121021c296178856ff484cb6805556190
332983b07673066592bca4398e3d43f4a49bffffffffff0228f307000000000001976a9140c8b4ce0b33d
041a3e114b659978fd2646774a9688aca00f0000000000001976a91454f82b54c82904451023ad3659
f448f2ba5d963d88ac00000000
```

sendrawtransaction 命令后面的参数为第七步结果中 hex 字段的值。

该命令的返回值如下：

```
0a739b9f33380ec7cb2eb3b9179928c2146b4fc3a54a7da1816ba004111b79db
```

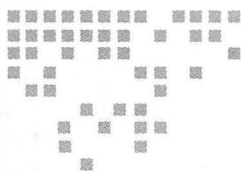
这是交易的地址，一般过 10 分钟左右交易完成。

## B.3 本章小结

本章主要介绍了比特币系统中 bitcoin-cli 模块的功能、子命令及其命令选项，熟悉并熟练使用 bitcoin-cli 模块对了解和操作比特币系统非常重要。







## Appendix C

### 附录 C

## 比特币系统的编程接口

比特币系统提供了基于 RESTAPI 的编程接口，这样能够突破语言的限制。任何语言只要能够支持 HTTP 协议，都能够无缝地和比特币系统完美结合。本章重点介绍比特币系统的 RESTAPI 接口提供的功能和调用方法。

### C.1 比特币 RESTAPI 接口的启动

#### C.1.1 快速启动一个 RESTAPI 的调用实例

bitcoind 模块启动的时候需要设置相关的参数才能顺利开启 RESTAPI 并给外部程序调用。在 bitcoind 模块中，以下参数是和 JSONRPC 接口密切相关的。

- rest：rest 有两个值，即 0 和 1，当 rest 设置为 1 的时候表示允许外部的 RESTAPI 请求，当值为 0 的时候表示不允许 RESTAPI 请求。
- rpcuser：RESTAPI 接口的账号。
- rpcpassword：RESTAPI 接口的密码。
- rpcport：RESTAPI 接口的端口号。
- rpcallowip：允许访问 RESTAPI 远程主机的 IP 地址。

bitcoind 模块中 RESTAPI 相关的参数可以通过配置文件或者命令选项的方式生效。

如果通过命令选项设置 RESTAPI 相关参数，则启动命令如下：

```
bitcoind -rest -rpcport=33133 -rpcuser=root -rpcpassword=111111 -rsdfkpcallowip=::/0
```



如果通过配置文件设置 RESTAPI 相关参数, 则配合文件需要添加如下内容:

```
rest=1
rpcuser=root
rpcpassword=111111
rpcport=33133
rsdfkpcallowip=::/0
```

以上配置只是启动 bitcoind 的 RESTAPI 服务的基本属性, 如果需要其他功能, 还要进行其他配置, 具体可以参考附录 B 相关内容。

配置完成之后启动 bitcoind 模块, 启动完成之后可以通过以下命令测试 RESTAPI 服务启动是否成功。

获取区块信息:

```
curl --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getblockchaininfo", "params": [] }' -H 'content-type: text/plain;' http://root:111111@127.0.0.1:33133/
```

获取网络信息:

```
curl --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getnetworkinfo", "params": [] }' -H 'content-type: text/plain;' http://root:111111@127.0.0.1:33133/
```

获取当前节点的钱包信息:

```
curl --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getwalletinfo", "params": [] }' -H 'content-type: text/plain;' http://root:111111@127.0.0.1:33133/
```

根据区块链高度获取相应区块的 Hash 值:

```
curl --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getblockhash", "params": [0] }' -H 'content-type: text/plain;' http://root:111111@127.0.0.1:33133/
```

获取区块详细信息:

```
curl --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getblock", "params": ["0000000000019d6689c085ael65831e934ff763ae46a2a6c172b3f1b60a8ce26f"] }' -H 'content-type: text/plain;' http://root:111111@127.0.0.1:33133/
```

如果上述信息执行成功, 则说明 bitcoind 模块 RESTAPI 接口已经顺利启动。

## C.1.2 RESTAPI 的请求参数和返回结果

通过上面的例子我们可以发现, 比特币的 RESTAPI 接口已提交给服务器的参数并被封装在一个 JSON 文件中, 返回的值的也是封装在一个 JSON 文件中。本节将详细介绍 JSON



文件的格式。

### 1. 请求参数的格式

bitcoind 模块 RESTAPI 接口的请求参数中包含的参数选项如表 C-1 所示。

表 C-1 bitcoind 模块 RESTAPI 接口请求接口参数表

参数名称	类型	是否必须	说明
jsonrpc	String	可选择	版本号信息
id	String	可选择	请求编号，返回信息中包含这个值
method	String	必须	请求方法
params	object	必须	请求方法的参数

### 2. 返回数据的格式

bitcoind 模块 RESTAPI 接口返回结果中包含的参数选项如表 C-2 所示。

表 C-2 bitcoind 模块 RESTAPI 接口返回结果参数表

参数名称	类型	是否必须	说明
result	any	必须	请求结果
error	object	必须	错误内容
code	number	必须	错误编号
message	string	必须	错误描述信息
id	string	必须	请求方法的 ID 值

返回结果中的 code 值可以在 bitcoind 的源码中获取详细的说明，相关源码的路径为 <https://github.com/bitcoin/bitcoin/blob/f914f1a746d7f91951c1da262a4a749dd3ebfa71/src/rpcprotocol.h>。

## C.2 通过 API 接口发起交易

RESTAPI 提供的功能和 bitcoin-cli 模块是完全一样的，只不过 RESTAPI 通常是提供给第三方的程序调用。本节将演示如果通过 RESTAPI 接口完成比特币交易。在执行下面的操作之前，请确保整个系统已经按照 A.5.1 节中的步骤顺利启动 bitcoind 模块并且已经同步了完整的交易数据。

第一步：获取历史记录。

```
curl --data-binary '{"jsonrpc": "1.0", "id": "mycurltest", "method":
```

```
"listunspent", "params": [] }' -H 'content-type: text/plain;' http://root:11111@192.168.23.228:33133/
```

## 第二步：创建交易。

```
curl --data-binary '{"jsonrpc": "1.0", "id": "mycurltest", "method": "createrawtransaction", "params": [{"[{"txid": "8396ba534fa59c01bfff84b6db41e1f4ec7c3ff5bc4e310334ca9b571b065f113", "vout": 0}], [{"15xc46B1PfqG6CWUTNfjTmqoUu2722qVmJ": 0.00526, "1L5Fmma1mHXcZH1cYJbWrrF93GQRJRhm78": 0.0003}]}]' -H 'content-type: text/plain;' http://root:11111@192.168.23.228:33133/
```

## 第三步：解锁钱包。

```
curl --user root:111111 --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "walletpassphrase", "params": ["a123456", 3600]}]' -H 'content-type: text/plain;' http://192.168.23.228:33133/
```

## 第四步：创建交易。

```
curl --data-binary '{"jsonrpc": "1.0", "id": "mycurltest", "method": "createrawtransaction", "params": [{"[{"txid": "8396ba534fa59c01bfff84b6db41e1f4ec7c3ff5bc4e310334ca9b571b065f113", "vout": 0}], [{"15xc46B1PfqG6CWUTNfjTmqoUu2722qVmJ": 0.00526, "1L5Fmma1mHXcZH1cYJbWrrF93GQRJRhm78": 0.0003}]}]' -H 'content-type: text/plain;' http://root:11111@192.168.23.228:33133/
```

## 第五步：解锁钱包准备转账。

```
curl --user root:111111 --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "walletpassphrase", "params": ["a123456", 3600]}]' -H 'content-type: text/plain;' http://192.168.23.228:33133/
```

## 第六步：对交易进行签名。

```
curl --user root:111111 --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "signrawtransaction", "params": ["020000000113f165b071b5a94c3310e3c45bffc3c74e1f1eb46d4bf8bf019ca54f53ba96830000000000ffffffffff02b0060800000000001976a9143664a45b9c256ac1c0b160472ce26c69baa6e0a088ac30750000000000001976a914d138627a09cfffcd006ef51c5c33e0464a46e52ad88ac00000000"]}]' -H 'content-type: text/plain;' http://192.168.23.228:33133/
```

## 第七步：发送交易。

```
curl --user root:111111 --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "sendrawtransaction", "params": ["020000000113f165b071b5a94c3310e3c45bffc3c74e1f1eb46d4bf8bf019ca54f53ba96830000000006a47304402200209f18fc3253f837e8650b91cb436dedea5a1687187bc9f3c921041ddf49fcf02203da625fd0016519e3a7a12a6f002208bd854ee870cd4c3815a49811bf10e26de012102d7f43819cdfd85bc80093255fc6f2fbd36b61e6aeadaaba2d725057f4b667f8effffffff02b0060800000000001976a9143664a45b9c256ac1c0b160472ce26c69baa6e0a088ac30750000000000001976a914d138627a09cfffcd006ef51c5c33e0464a46e52ad88"]}]'
```



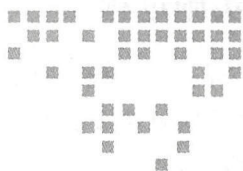


```
ac00000000"] }' -H 'content-type: text/plain;' http://192.168.23.228:33133/
```

比特币的 RESTAPI 接口使得任何编程语言只要支持 HTTP 协议都可以非常简单地访问比特币系统，从而有效避免了客户端因编程语言多样性带来的各种问题。

### C.3 本章小结

本章主要介绍了比特币 RESTAPI 接口的功能特性和调用方法，比特币 RESTAPI 接口是比特币系统提供的对外接口，熟悉比特币 RESTAPI 接口的使用方法有助于开发基于比特币的第三方应用。



## 比特币系统客户端项目实战

本章将创建一个简单的比特币系统的客户端应用，通过阅读本章内容，读者可以了解如何通过比特币系统提供的 JSON-RPC 接口开发应用程序。

### D.1 项目背景

比特币系统中的 bitcoin-cli 模块提供的基于命令行的接口可用于操作比特币系统，但是命令行接口只能在本机操作比特币系统，而且需要一定的专业技能，最关键的是命令的接口无法方便地和第三方应用结合。bitcoind 模块提供了 JSON-RPC 接口，通过这些 JSON-RPC 接口，只有能支持 HTTP 协议的编程语言都和非常容易的和比特币系统进行交互。本例将基于 bitcoind 模块的 JSON-RPC 接口开发一个基于浏览器的 Web 应用程序，通过这个 Web 应用只要通过浏览器即可完成相关对比特币系统的相关操作。

限于本书篇幅的限制本例仅仅选取几个典型的操作场景，这些操作场景如下：

- 获取当前比特币系统的区块信息。
- 获取当前比特币系统的网络信息。
- 获取当前钱包信息。
- 根据区块的高度获取区块链的 Hash 值。
- 根据区块的 Hash 值，获取区块链的详细信息。

为了和前面的示例统一，本例依然采用 Nodejs 作为开发语言。

## D.2 项目实施过程

运行本例之前首先请按照附录 A 的内容安装并且启动了一个比特币系统。注意：在启动的过程中需要按照附录 A 的内容打开比特币系统的 JSONRPC 接口。启动完成之后按照下列步骤完成示例项目。

### 1. 项目准备

在本例中会用 Nodejs 的 express 框架作为 Web 框架，因此在项目开始之前需要通过 npm 工具安装 express 框架的相关包。安装前先进入项目的目录，然后执行以下安装命令：

```
npm install express
```

执行完成之后，express 模块的相关依赖包会被安装在项目文件中的 node\_modules 文件夹中。

### 2. 比特币 JSONRPC 接口的封装

比特币的 JSONRPC 接口的调用比较简单，我们将相关的操作封装在一个接口中，这样更便于调用。我们将这个接口的源代码文件命名为 bitcoindservice.js，接口的源代码如下：

```
var http=require('http');
var querystring=require('querystring');
var url = require('url')

var bithttppost = function (posturl,port,postData,username,passwd) {

    var postDatastr=JSON.stringify(postData);

    var urlObj = url.parse(posturl)

    var loginstring = username + ":" + passwd;

    var loginstringbuf = new Buffer( loginstring );
    var cred = loginstringbuf.toString('base64');

    var options={
        hostname:urlObj.hostname,
        port:port,
        path: urlObj.pathname,
        method:'POST',
        headers:{

            'Content-Type':'text/plain',
            'Content-Length':Buffer.byteLength(postDatastr),
```

```

        'Authorization': `Basic ${cred}`
    }
}

return httpPost(options, postDataStr);
}

var httpPost = function (options, postData) {
    return new Promise((resolve, reject) => {
        var buffers = [];
        var req = http.request(options, function (res) {
            res.on('data', function (reposebuffer) {
                buffers.push(reposebuffer);
            });
            res.on('end', function () {
                var wholeData = Buffer.concat(buffers);
                var dataStr = wholeData.toString('utf8');
                resolve(dataStr);
            });
            res.on('error', function (err) {
                reject(err);
            });
        });
        req.write(postData);
        req.end();
    });
}

exports.httpPost = httpPost;
exports.bitHttpPost = bitHttpPost;

```

上述代码封装了对比特币系统 JSON-RPC API 的底层操作，如需要使用，则直接引入该模块并调用模块的相关方法即可。



### 3. Web 界面相关的代码

现在可以开发 Web 服务相关的代码了, 我们将 Web 服务的相关源代码保存在名为 bitmain.js 的文件中。代码如下:

```
var co = require('co');
var bitcoindservice = require('./bitcoindservice');
var express = require('express');

var app = express();

var bitcoind_host = "http://192.168.23.212";
var bitcoind_port = 33133;
var bitcoind_username = "root";
var bitcoind_passwd = "111111";

// 获取当前比特币系统的区块信息

app.get('/getblockchaininfo', function (req, res) {

  co( function * () {

    var bitcommand = {"jsonrpc": "1.0", "id": "curltest", "method":
"getblockchaininfo", "params": [] };
    let content = yield bitcoindservice.bithttppost(bitcoind_host, bitcoind_
port, bitcommand ,bitcoind_username,bitcoind_passwd);
    res.send( content );

  }).catch((err) => {
    res.send(err);
  })

});

// 获取当前比特币系统的网络信息

app.get('/getnetworkinfo', function (req, res) {

  co( function * () {

    var bitcommand = {"jsonrpc": "1.0", "id": "curltest", "method":
"getnetworkinfo", "params": [] };
    let content = yield bitcoindservice.bithttppost(bitcoind_host, bitcoind_
port , bitcommand ,bitcoind_username,bitcoind_passwd);
    res.send( content );

  }).catch((err) => {
```

```
        res.send(err);
    })

});

// 获取当前的钱包信息
app.get('/getwalletinfo', function (req, res) {

    co( function * () {

        var bitcommand = {"jsonrpc": "1.0", "id": "curltest", "method":
"getwalletinfo", "params": [] };
        let content = yield bitcoindservice.bithttppost(bitcoind_host, bitcoind_
port , bitcommand ,bitcoind_username,bitcoind_passwd);
        res.send( content );

    }).catch((err) => {
        res.send(err);
    })

});

// 根据区块的高度获取区块链的 Hash 值
app.get('/getblockhash', function (req, res) {

    co( function * () {

        var bitcommand = {"jsonrpc": "1.0", "id": "curltest", "method":
"getblockhash", "params": [0] };
        let content = yield bitcoindservice.bithttppost(bitcoind_host, bitcoind_
port , bitcommand ,bitcoind_username,bitcoind_passwd);
        res.send( content );

    }).catch((err) => {
        res.send(err);
    })

});

// 根据区块的 Hash 值, 获取区块链的详细信息
app.get('/getblock', function (req, res) {
```

## 302 区块链开发实战: Hyperledger Fabric 关键技术与案例分析

```

co( function * () {

    var bitcommand = {"jsonrpc": "1.0", "id": "curltest", "method": "getblock",
"params": ["000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f"] };
    let content = yield bitcoindservice.bithttppost(bitcoind_host, bitcoind_
port , bitcommand ,bitcoind_username,bitcoind_passwd);
    res.send( content );

    }).catch((err) => {
        res.send(err);
    })

});

// 启动 HTTP 服务
var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;

    console.log('Example app listening at http://%s:%s', host, port);
});

// 注册异常处理器
process.on('unhandledRejection', function (err) {
    console.error(err.stack);
});

process.on('uncaughtException', console.error);

```

现在可以启动这个简单的客户端程序了, 启动命令如下:

```
node bitmain.js
```

启动成功之后可以在浏览器中输入以下网址进行相关的操作, 相关功能的请求地址如下所示:

```
// 获取当前比特币系统的区块信息
http://localhost:3000/getblockchaininfo
```

```
// 获取当前比特币系统的网络信息
http://localhost:3000/getnetworkinfo
```

```
// 获取当前钱包信息
```

```
http://localhost:3000/getwalletinfo
```

```
// 根据区块的高度获取区块链的 Hash 值
```

```
http://localhost:3000/getblockhash
```

```
// 根据区块的 Hash 值，获取区块链的详细信息
```

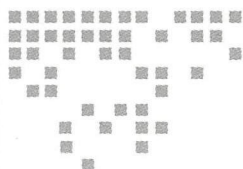
```
http://localhost:3000/getblock
```

至此，一个简单的 bitcoind 管理客户端就开发完成了。

## D.3 本章小结

通过对本章内容的学习，读者可以熟悉如何通比特币系统提供的 JSONRPC 接口在第三方应用程序中调用比特币系统的相关功能。虽然本例使用 Nodejs 语言开发，但是因为 HTTP 协议是使用非常广泛的协议，几乎所有语言对其都提供了很好的支持，所以使用其他语言进行开发也非常简单。





## Appendix E

### 附录 E

# 区块链相关术语

- **Block (区块)**: 在区块链网络上承载永久记录数据的包。
- **Blockchain (区块链)**: 一个共享的分布式账本，其中交易通过附加块永久记录。区块链作为所有交易的历史记录，从发生块到最新的块都包含其中，因此命名为 blockchain (区块链)。
- **Block Height (区块高度)**: 连接在区块链上的块数。
- **Block Reward (积分奖励)**: 它是对在采矿期间成功计算区块中 Hash 的矿工的一种激励形式。在区块链上进行交易验证的过程中会产生新的币，矿工将获得其中的一部分作为奖励。
- **Central Ledger (中央账簿)**: 由中央机构维持的分类账。
- **Confirmation (确认)**: 去中心化的一次交易。
- **Consensus (共识)**: 当所有网络参与者同意交易的有效性时，可达成共识，确保分布式账本是彼此的精确副本。
- **Cryptocurrency (加密货币)**: 也称为令牌，加密货币是数字资产的呈现方式。
- **Cryptographic Hash Function (加密哈希函数)**: 密码 Hash 产生从可变大小交易输入固定大小和唯一 Hash 值。SHA-256 算法是加密散列的一个例子。
- **DApp (去中心化应用)**: DApp (去中心化应用程序) 是一种开源的应用程序，自动运行，将其数据存储存储在区块链上，以密码令牌的形式激励，并以显示有价值证明的协议进行操作。
- **DAO (去中心化自治组织)**: 去中心化自治组织可以被认为是在没有任何人为干预的

情况下运行的公司，并将一切形式的控制权交给一套不可破坏的业务规则。

- **Distributed Ledger (分布式账本)**: 数据通过分布式节点网络进行存储。分布式账本不是必须具有自己的货币，它可能会被许可和私有。
- **Distributed Network (分布式网络)**: 处理能力和数据分布在节点上，且没有集中式数据中心的一种网络。
- **Difficulty (容易程度)**: 指成功挖掘交易信息的数据块的难易程度。
- **Digital Signature (数字加密)**: 通过公钥加密生成的数字代码，附加到电子传输的文档以验证其内容和发件人的身份。
- **Double Spending (双重支付)**: 当花费一笔钱多于一次支付限额时，就会发生双重支付。
- **Genesis Block (创世区块)**: 区块链上的第一个区块。
- **Hard Fork (硬分支)**: 一种使以前无效的交易变为有效的分支类型，反之亦然。这种类型的分支需要所有节点和用户升级到最新版本的协议软件。
- **Hash (哈希) 算法**: 将任意长度的输入值映射为较短的固定长度的二进制值。数据的 Hash 值可以检验数据的完整性，一般用于快速查找和加密算法。Hash 算法广泛应用于区块链中，比如 Merkle 树、以太坊账户地址、比特币地址、POW 算法等。
- **Hybrid POS/POW (混合 POS / POW)**: POW (Proof of Work, 工作证明) 是指获得多少货币，取决于挖矿贡献的工作量，电脑性能越好，分得的矿就越多。POS (Proof of Stake, 股权证明) 根据持有货币的量和时间进行利息分配的制度，在 POS 模式下，你的“挖矿”收益正比于你的币龄，而与电脑的计算性能无关。混合 POS/POW 可以将网络上的共享分发算法作为共享证明和工作证明。在这种方法中，可以实现矿工和选民（持有者）之间的平衡，由内部人（持有人）和外部人（矿工）创建一个基于社区的治理体系。
- **Merkle Tree (默克尔树)**: Merkle 树在分布式环境下进行验证、文件对比时应用较多。区块链系统采用二叉树型的 Merkle 树对这些交易进行归纳表示，同时生成该交易集合的数字签名。Merkle 树支持快速归纳和校验区块中交易的完整性与存在性。
- **Multi-Signature (多重签名)**: 多重签名，可以简单理解为一个数字资产的多个签名。多重签名预示着数字资产可由多人支配和管理。在加密货币领域，如果要动用一個加密货币地址的资金，通常需要该地址的所有人使用他的私钥（由用户专属保护）进行签名。那么多重签名，就是动用这笔资金需要多个私钥签名，通常这笔资金或数字资产会保存在一个多重签名的地址或账号里。
- **Node (节点)**: 由区块链网络的参与者操作的分类账的副本。
- **Oracle (预言机)**: 向智能合约提供数据，它是现实世界和区块链之间的桥梁。

- **Public Address (公用地址)**: 即公钥的密码 Hash 值, 其是可以在任何地方发布的电子邮件地址, 与私钥不同。
- **Private Key (私钥)**: 是一串数据, 它允许用户访问特定钱包中的令牌。它们作为密码, 除了地址的所有者之外, 其他人均不可见。
- **POW (工作量证明)**: 比特币系统利用 POW 机制使系统各节点最终达成共识, 进而得到最终区块。这里的工作是指找到一个合理的区块 Hash 值, 需要不断进行大量计算。
- **POS (股权证明)**: 权益证明机制, 这种机制根据货币持有量和时间来分配相应的利息, 不足的地方是因为没有消耗大量算力导致货币价值来源难以确定。
- **Script**: 一种 Litecoin 使用的加密算法。与 SHA256 相比, 它的速度更快, 因为它不会占用很多处理时间。
- **SHA-256**: 比特币等数字货币使用的加密算法。然而, 它使用了大量的计算能力和处理时间, 迫使矿工组建采矿池以获取收益。
- **Smart Contracts (智能合约)**: 智能合约将可编程语言的业务规则编码到区块上, 并由网络的参与者实施。
- **Soft Fork (软分支)**: 软分支与硬分支不同之处在于, 只有先前有效的交易才能使其无效。由于旧节点将新的块识别为有效, 所以软分支基本上是向后兼容的。这种分支需要大多数矿工升级才能执行, 而硬分支需要所有节点与新版本达成一致。
- **Testnet**: 开发商使用的测试区块链, 它主要是用来防止改变主链上的资产。
- **Transaction Block (交易区块)**: 聚集到一个块中的交易的集合, 可以将其散列并添加到区块链中。
- **Transaction Fee (手续费)**: 所有的加密货币交易都会涉及一笔很小的手续费。向以太坊网络发送交易是需要附带一笔交易费用, 这笔费用是矿工帮助网络打包交易的奖励。
- **Turing Complete (图灵完备)**: 指机器执行任何其他可编程计算机能够执行计算的能力, 比如 Ethereum 虚拟机 (EVM)。
- **Wallet (钱包)**: 一个包含私钥的文件。它通常包含一个软件客户端, 允许访问、查看和创建钱包所涉及的特定块链的交易。



## 作者简介

**冯翔** 资深区块链技术专家，IONChain（离子链）CTO，上海旺链科技区块链研究院负责人，Hyperledger 核心项目核心代码开发者。中国区块链技术的早期探索者和传播者，创立了有广泛影响力的区块链技术社区“区块链兄弟”。已经参与过多个基于区块链技术的落地项目，现阶段主要致力于区块链技术和传统行业的融合，尤其关注区块链技术和物联网技术的结合。

**刘涛** 上海旺链信息科技有限公司 CEO，IONChain（离子链）创始人，致力于结合中国本土情况的区块链研究开发，现为复旦大学区块链研究生课程讲师。前埃森哲高级总监，在高科技制造、汽车、金融行业有超过 15 年的业务咨询和技术架构经验，曾担任华为、Alcatel-lucent、上汽通用、平安、中国移动高端外部顾问。

**吴寿鹤** 资深区块链技术专家，IONChain（离子链）首席架构师，HyperLedger 核心项目开发人员，同时对以太坊相关技术有深入研究。是国内区块链技术领域的早期实践者和布道者，一直积极推动区块链技术的传播和应用落地，是国内知名区块链技术社区“区块链兄弟”的联合创始人，目前从事基于物联网的区块链底层基础平台的开发。个人主页：<http://gcc2ge.github.io>。

**周广益** 上海指旺金科 CEO，中国区块链技术的早期探索者，36Kr、未央网、共享财经等媒体特约作家，现为复旦大学软件学院特聘讲师。



这是一本强调实战的书，也是目前区块链社区比较缺乏的书，本书的两位作者都是超级账本的开发。我相信这本书能够很好地帮助读者快速掌握 Hyperledger Fabric 的开发技能，我推荐您阅读这本书。

——Julian Goldon 超级账本亚太区副总裁

Hyperledger Fabric 是联盟链领域的重要平台，为许多联盟链项目所采用，已成为联盟链开发事实上的首选平台。这一平台不仅实现了不含原生代币的基础账本和智能合约，而且在集约化的 BaaS 服务支持方面也做了大量铺垫，可以方便地在同一批硬件基础设施上为小型企业虚拟出不同的专用区块链来。

这是一本介绍如何在 Hyperledger Fabric 上进行应用开发的工具书。从一个程序员的视角，将在 Hyperledger Fabric 上从事开发工作必备的知识和技能做了系统性介绍，并附有大量实例。全书基础扎实、内容实用，适合区块链的企业 / 行业级应用开发者学习参考。

——白硕 ChinaLedger 技术委员会主任 / 中科院博士生导师

联盟链适用于很多行业，有很多应用场景，Hyperledger Fabric 技术已经成为联盟链开发的事实标准。本书理论与实战兼顾，专为没有区块链开发基础的读者量身打造，首先从理论角度讲解了 Hyperledger Fabric 的基本概念、实现原理、关键技术，然后讲解了如何基于 Hyperledger Fabric 开发应用，最后是多个相关的实战案例，循序渐进，通俗易懂。通过本书，相信读者能迅速掌握 Hyperledger Fabric 的相关技术和应用开发方法。

——李庆华 MATRIX 链 CTO

本书理论与实践相结合，首先讲解了 Hyperledger Fabric 的架构、原理和核心技术，然后介绍了应用开发的方法，最后结合多个具体案例展示了如何应用 Hyperledger Fabric 开发区块链应用系统。内容翔实具体、深入浅出，更令人惊喜的是，读者从本书中不仅能学到如何使用区块链应用开发，还能品味到作者解决问题的技巧和对区块链的深入思考，是学习区块链应用开发实战的精品。

——钱汉涛 阿希链 CTO

以太坊已成为公链技术最具竞争力的开放标准之一，活跃的开发者社区以及相对丰富的 DApp 类型促成了良好的应用生态，以太坊也极大地推动了区块链的发展。而以超级账本为代表的 DLTs（分布式账本技术）则从企业层面补足了公链技术的一些不足，在受限环境中把分布式系统的优势发挥到极致。以太坊和超级账本技术的应用和发展，离不开像本书的 4 位作者这样的布道者和贡献者，他们无私地将自己在实践中总结出来的经验呈现在了这两本书中，对于区块链开发的初学者而言，具有很重要的学习和参考价值。

——陈浩 元界 CTO



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机/程序设计

ISBN 978-7-111-59942-5



定价: 79.00元